

Оглавление

1.	Масштабируемая Веб-архитектура и распределенные системы	10
1.1	Принципы построения распределенных веб-систем	10
1.2	Основы	11
1.3.	Структурные компоненты быстрого и масштабируемого доступа к данным	18
1.4.	Заключение	29
2.	Технология подготовки релизов веб-браузера Firefox.....	29
2.1.	Рассмотрите N способов до момента подготовки релиза	30
2.2.	"Отправка на сборку".....	32
2.3.	Использование тэгов, сборка и архивы с исходным кодом	34
2.4.	Повторно упакованные сборки с локализацией и партнерские повторно упакованные сборки	37
2.5.	Добавление цифровых подписей.....	39
2.6.	Обновления.....	41
2.7.	Размещение сборок на внутренних зеркалах и контроль качества	44
2.8.	Перемещение файлов на публичные зеркала и в систему автоматического обновления	44
2.9.	Выученные уроки.....	45
2.10.	Дополнительная информация	47
3.	ОС реального времени FreeRTOS	47
3.1.	Что такое «встроенная» и «реального времени»?	48
3.2.	Обзор архитектуры	48
Об аппаратном обеспечении	49	
3.3.	Планирование задач: Краткий обзор.....	51
Приоритеты задач и список готовности	51	
Тактовая частота системы	51	
3.4.	Задачи	52
Блок управления задачей TCB	52	
Настройка задачи	54	
3.5.	Списки	55
3.6.	Очереди	60
Семафоры и мютексы	62	
Реализация	62	
3.7.	Заключение	63
3.8.	Благодарности	63
4.	GDB	64
4.1.	Цель разработки	64
4.2.	Начало развития GDB.....	65
4.3.	Блочная диаграмма	65
4.4.	Примеры работы	66

4.5. Переносимость	67
4.6. Структуры данных	68
4.7. Часть, ответственная за работу с символами	70
4.8. Часть, ответственная за работу в целевой системе.....	71
Gdbarch.....	72
4.9. Интерфейсы GDB.....	76
4.10. Процесс разработки	78
4.11. Выученные уроки.....	78
5. Компилятор Glasgow Haskell	80
5.1. Что такое язык Haskell?	81
5.2. Общий взгляд на проект.....	82
Оцениваем размер кода	83
Компилятор.....	84
Компиляция кода на языке Haskell	84
Синтаксический разбор	85
Переименование	86
Проверка типов	86
Сокращение синтаксического разнообразия языка и язык Core	87
Оптимизация.....	87
Генерация кода.....	88
5.3. Ключевые проектные решения	88
Промежуточный язык.....	88
Проверка типов исходного языка.....	90
Без таблиц символов	91
Межмодульная оптимизация	92
5.4. Средства расширяемости	93
Правила преобразований, определяемые пользователями	94
Плагины компилятора	94
Компилятор GHC как библиотека: интерфейс API компилятора GHC.....	95
Система пакетов	96
5.5. Система времени выполнения	97
Ключевые проектные решения	97
5.6. Разработка компилятора GHC	100
Комментарии и замечания	100
Как поддерживать рефакторинг	101
Отступления от правил.....	102
Разработка системы RTS	103
Борьба со сложностью	104
Инварианты и их проверка.....	104
5.7. Заключение	105

6.	Git.....	105
6.1.	Пара слов о Git	106
6.2.	Начало развития проекта Git.....	106
6.3.	Архитектура системы контроля версий.....	107
6.4.	Тулкит	110
6.5.	Репозиторий, данные индексирования и рабочие области	111
6.6.	База данных объектов.....	113
6.7.	Техники хранения и сжатия данных	114
6.8.	История объединения ветвей	115
6.9.	Что дальше?	117
6.10.	Выученные уроки.....	118
7.	Проект GPSD	119
7.1.	Почему существует проект GPSD	119
7.2.	Взгляд извне	120
7.3.	Слои программного обеспечения	121
7.4.	Поток данных	124
7.5.	Защищая архитектуру.....	125
7.6.	Нет конфигурирования — нет суэты	126
7.7.	Ограничения, имеющиеся во встроенных системах, полезны	127
7.8.	JSON и архитектонауты.....	128
7.9.	Проектирование с минимизацией количества дефектов.....	129
7.10.	Усвоенные уроки.....	130
8.	Среда времени выполнения динамических языков и языки Iron	132
8.1.	История	133
8.2.	Принципы реализации среды времени выполнения динамических языков	134
8.3.	Особенности реализации языков	135
8.4.	Синтаксический анализ	135
8.5.	Деревья выражений	136
8.6.	Интерпретация и компиляция.....	137
8.7.	Точки динамических вызовов.....	138
8.8.	Протокол метаобъектов.....	141
8.9.	Хост-среда.....	144
8.10.	Общая структура	146
8.11.	Усвоенные уроки.....	146
9.	ITK	147
9.1.	Что такое ИТК?.....	147
9.2.	Возможности архитектуры	147
	Характер зверя.....	147
	Модульность.....	149
	Конвейер данных	152

Объекты процесса и данных	153
Иерархия классов конвейера.....	154
Внутренние процессы конвейера	155
Фабрики	156
Потоковая передача данных	159
9.3. Выученные уроки.....	162
Повторное использование	162
Обобщенное программирование	162
Знать, когда остановиться	163
Возможность поддержки кода	164
Невидимая рука	164
Рефакторинг.....	165
Воспроизводимость	166
10. GNU Mailman.....	167
10.1. Анатомия сообщения.....	168
10.2. Список рассылки.....	171
10.3. Обработчики.....	173
10.4 Ведущий обработчик	175
10.5. Правила, звенья и цепочки.....	176
10.6. Обработчики сообщений и каналы	179
10.7. VERP	181
10.8. REST	182
10.9. Интернационализация	183
10.10. Выученные уроки.....	184
11. Библиотека matplotlib	185
11.1. Проблема ключа аппаратной защиты	185
11.2. Обзор архитектуры библиотеки matplotlib	186
11.3. Рефакторинг системы вывода данных	192
11.4. Преобразования.....	193
11.5. Процесс обработки полилиний	195
11.6. Математические выражения	196
11.7. Тестирование с целью поиска регрессий.....	197
11.8. Выученные уроки.....	198
12. MediaWiki	199
12.1. Исторический обзор.....	200
12.2. Кодовая база и практика разработки приложения MediaWiki.....	202
12.3. База данных и хранилище текста	204
12.4. Запросы, кэширование и доставка данных	206
12.5. Языки.....	209
12.6. Пользователи	211

12.7. Содержимое статей	212
12.8. Модификации и расширение возможностей MediaWiki	215
12.9. Планы на будущее.....	217
12.10. Материалы для дополнительного чтения	217
12.11. Благодарности	218
13. Moodle	218
13.1. Обзор принципа работы приложения Moodle	219
13.2. Система ролей и разрешений приложения Moodle	223
13.3. Вернемся к нашему примеру сценария.....	226
13.4. Генерация результирующего документа	228
13.5. Абстракция для работы с базой данных	231
13.6. Что не было описано.....	233
13.7. Выученные уроки.....	233
14. NGINX.....	234
14.1. Почему высокоэффективные параллельные вычисления так важны?	234
Apache не подходит?.....	235
Есть ли ещё преимущества при использовании nginx?	236
14.2. Обзор архитектуры Nginx	237
Структура исходного кода	238
Модель исполнителя.....	239
Назначение процессов nginx	240
Краткий обзор кеширования в nginx	241
14.3 Настройки nginx	241
14.4. Внутреннее устройство nginx	243
14.5. Выводы.....	248
15. Open MPI.....	248
15.1. Введение	248
Интерфейс передачи сообщений (MPI)	249
Использование MPI.....	249
Open MPI.....	250
15.2. Архитектура.....	251
Архитектура слоев абстракции.....	251
Архитектура плагинов	253
Фреймворки плагинов	255
Фреймворки типа «несколько из нескольких».....	256
Фреймворки типа «один из нескольких».....	256
Динамические фреймворки.....	256
Статические фреймворки	256
Компоненты плагинов	257
Структура компонента.....	257

Структура модуля	259
Объединяем все вместе	260
Параметры времени выполнения	261
15.3. Усвоенные уроки.....	262
Производительность	262
Усвоенный урок:	262
Стоя на плечах гигантов.....	262
Усвоенный урок:	263
Оптимизация обычно выполняемых операций	263
Усвоенный урок:	263
Прочие уроки.....	263
Заключение	264
16. OSCAR	264
16.1. Системная иерархия.....	265
16.2. Принятые в прошлом решения	265
16.3. Управление версиями	266
16.4. Модели данных/DAO.....	266
16.5. Права доступа.....	270
16.6. Интегратор	272
16.6. Интегратор	273
16.7. Выученные уроки.....	276
17. Processing.js.....	278
17.1. Как это работает?	279
17.2. Значительные различия	280
17.3. Компоненты кода	288
17.4. Разработка библиотеки Processing.js	291
17.5. Выученные уроки.....	294
18. Puppet.....	295
18.1. Введение	295
18.2. Обзор архитектуры	298
18.3. Анализ компонентов.....	300
18.4. Инфраструктура	306
18.5. Выученные уроки.....	308
18.6. Заключение	308
19. PyPy	309
19.1. Немного истории.....	309
19.2. Обзор PyPy.....	309
19.3. Интерпретатор языка Python.....	310
19.4. Инструменты преобразования кода для языка RPython.....	311
19.5. JIT-компиляция в PyPy	316

19.6. Недостатки архитектуры	319
19.7. Немного о процессе разработки	320
19.8. Резюме.....	321
19.9. Выученные уроки.....	322
20. SQLAlchemy	322
20.1. Сложность создания слоя абстракции для баз данных	323
20.2. Дихотомия между основными задачами и объектно-реляционным отображением	324
20.3. Использование DBAPI.....	326
20.4. Описание схемы	330
20.5. SQL-запросы.....	332
20.6. Отображение классов при использовании объектно-реляционного отображения.....	335
20.7. Методы выполнения запросов и загрузки данных	339
20.8. Сессия и индивидуальное отображение	341
20.9. Рабочая единица.....	345
20.10. Заключение	349
21. Twisted.....	350
21.1. Почему Twisted?.....	350
21.2. Архитектура Twisted.....	352
Повторное использование существующих приложений.....	354
Шаблон проектирования reactor	354
Управление цепочками функций обратного вызова	355
Объекты Deferred	357
Транспорты.....	359
Протоколы	359
Приложения.....	360
Служба	360
Приложение	361
Файлы ТАС	361
twistd.....	362
Плагины	362
21.3. Взгляд в прошлое и выученные уроки.....	364
22. Фреймворк Yesod	366
22.1. Сравнение с другими фреймворками.....	367
22.2. Интерфейс веб-приложений.....	368
Типы данных	369
Потоки.....	369
Сборщик.....	370
Обработчики	370
Промежуточный слой middleware	372
Тесты wai-test.....	372

22.3. Шаблоны	372
Типы	373
Другие языки	374
22.4. Хранение данных с помощью Persistent	375
Терминология	375
Безопасность типов	376
Межбазовый синтаксис	377
Миграция	377
Отношения	377
22.5. Yesod	378
Маршруты	379
Адреса типобезопасных URL	380
Обработчики	380
Виджеты	381
Подсайты subsite	382
22.6. Усвоенные уроки	382
23. Проект Yocto	383
23.1. Введение в систему сборки Poky Build System	384
Концепции системы сборки Poky	386
23.2. Архитектура BitBake	389
Механизм межпроцессного взаимодействия системы BitBake	389
Хранилище данных DataSmart системы BitBake с возможностью копирования при записи ..	391
Планировщик BitBake	392
Зависимости	393
Очередь выполнения сборки	394
23.3. Заключение	394
23.4. Благодарности	396
24.1. Приложение или библиотека	396
24.2. Глобальное состояние	397
24.3. Производительность	398
24.4. Критический путь	399
24.5. Выделение памяти	400
24.6. Пакетная обработка	401
24.7. Общий обзор архитектуры	403
24.8. Модель распараллеливания	404
24.9. Неблокирующие алгоритмы	406
24.10. Интерфейс API	407
24.11. Шаблоны обмена сообщениями	408
24.12. Заключение	410
24. Система обмена сообщениями ZeroMQ	410

24.1. Приложение или библиотека	411
24.2. Глобальное состояние.....	412
24.3. Производительность	413
24.4. Критический путь	415
24.5. Выделение памяти	415
24.6. Пакетная обработка	416
24.7. Общий обзор архитектуры	418
24.8. Модель распараллеливания.....	420
24.9. Неблокирующие алгоритмы	421
24.10. Интерфейс API	423
24.11. Шаблоны обмена сообщениями	424
24.12. Заключение	426

1. Масштабируемая Веб-архитектура и распределенные системы

Открытое программное обеспечение стало основным структурным элементом при создании некоторых крупнейших веб-сайтов. С ростом этих веб-сайтов возникли передовые практические методы и руководящие принципы их архитектуры. Данная глава стремится охватить некоторые ключевые вопросы, которые следует учитывать при проектировании больших веб-сайтов, а также некоторые базовые компоненты, используемые для достижения этих целей.

Основное внимание в данной главе уделяется анализу веб-систем, хотя часть материала может быть экстраполирована и на другие распределенные системы

1.1 Принципы построения распределенных веб-систем

Что именно означает создание и управление масштабируемым веб-сайтом или приложением? На примитивном уровне это просто соединение пользователей с удаленными ресурсами через Интернет. А ресурсы или доступ к этим ресурсам, которые рассредоточены на множестве серверов и являются звеном, обеспечивающим масштабируемость веб-сайта.

Как большинство вещей в жизни, время, потраченное заранее на планирование построения веб-службы, может помочь в дальнейшем; понимание некоторых соображений и компромиссов, стоящих позади больших веб-сайтов, может принести плоды в виде более умных решений при создании меньших веб-сайтов. Ниже некоторые ключевые принципы, влияющие на проектирование крупномасштабных веб-систем:

- **Доступность:** длительность работоспособного состояния веб-сайта критически важна по отношению к репутации и функциональности многих компаний. Для некоторых более крупных онлайновых розничных магазинов, недоступность даже в течение нескольких минут может привести к тысячам или миллионам долларов потерянного дохода. Таким образом, разработка постоянно доступных и устойчивых к отказам систем является фундаментальным деловым и технологическим требованием. Высокая доступность в распределенных системах требует внимательного рассмотрения избыточности для ключевых компонентов, быстрого восстановления после частичных системных отказов и сглаженного сокращения возможностей при возникновении проблем.
- **Производительность:** Производительность веб-сайта стала важным показателем для большинства сайтов. Скорость веб-сайта влияет на работу и удовлетворенность пользователей, а также ранжирование поисковыми системами - фактор, который непосредственно влияет на удержание аудитории и доход. В результате, ключом является создание системы, которая оптимизирована для быстрых ответов и низких задержек.
- **Надежность:** система должна быть надежной, таким образом, чтобы определенный запрос на получение данных единообразно возвращал определенные данные. В случае изменения данных или обновления, то тот же запрос должен возвращать новые данные. Пользователи должны знать, если что-то записано в систему или храниться в ней, то можно быть уверенным, что оно будет оставаться на своем месте для возможности извлечения данных впоследствии.
- **Масштабируемость:** Когда дело доходит до любой крупной распределенной системы, размер оказывается всего лишь одним пунктом из целого списка, который необходимо учитывать. Не менее важным являются усилия, направленные на увеличение пропускной способности для обработки больших объемов нагрузки, которая обычно и называется масштабируемостью системы. Масштабируемость может относиться к различным параметрам сис-

темы: количество дополнительного трафика, с которым она может справиться, насколько легко нарастить ёмкость запоминающего устройства, или насколько больше других транзакций может быть обработано.

- **Управляемость:** проектирование системы, которая проста в эксплуатации еще один важный фактор. Управляемость системы приравнивается к масштабируемости операций "обслуживание" и "обновления". Для обеспечения управляемости необходимо рассмотреть вопросы простоты диагностики и понимания возникающих проблем, легкости проведения обновлений или модификации, прихотливости системы в эксплуатации. (То есть, работает ли она как положено без отказов или исключений?)
- **Стоимость:** Стоимость является важным фактором. Она, очевидно, может включать в себя расходы на аппаратное и программное обеспечение, однако важно также рассматривать другие аспекты, необходимые для развертывания и поддержания системы. Количество времени разработчиков, требуемое для построения системы, объем оперативных усилий, необходимые для запуска системы, и даже достаточный уровень обучения - все должно быть предусмотрено. Стоимость представляет собой общую стоимость владения.

Каждый из этих принципов является основой для принятия решений в проектировании распределенной веб-архитектуры. Тем не менее, они также могут находиться в противоречии друг с другом, потому что достижение целей одного происходит за счет пренебрежения другими. Простой пример: выбор простого добавления нескольких серверов в качестве решения производительности (масштабируемость) может увеличивать затраты на управляемость (вы должны эксплуатировать дополнительный сервер) и покупку серверов.

При разработке любого вида веб-приложения важно рассмотреть эти ключевые принципы, даже если это должно подтвердить, что проект может пожертвовать одним или несколькими из них.

1.2 Основы

При рассмотрении архитектуры системы есть несколько вопросов, которые необходимо осветить, например: какие компоненты стоит использовать, как они совмещаются друг с другом, и на какие компромиссы можно пойти. Вложение денег в масштабирование без очевидной необходимости в ней не может считаться разумным деловым решением. Однако, некоторая предусмотрительность в планировании может существенно сэкономить время и ресурсы в будущем.

Данный раздел посвящается некоторым базовым факторам, которые являются важнейшими для почти всех больших веб-приложений: *сервисы, избыточность, сегментирование, и обработка отказов*. Каждый из этих факторов предполагает выбор и компромиссы, особенно в контексте принципов, описанных в предыдущем разделе. Для пояснения приведем пример.

Пример: Приложение хостинга изображений

Вы, вероятно, когда-либо уже размещали изображения в сети. Для больших сайтов, которые обеспечивают хранение и доставку множества изображений, есть проблемы в создании экономически эффективной, высоконадежной архитектуры, которая характеризуется низкими задержками ответов (быстрое извлечение).

Вообразите систему, где пользователи имеют возможность загрузить свои изображения на центральный сервер, и при этом изображения могут запрашиваться через ссылку на сайт или API, аналогично Flickr или Picasa. Для упрощения описания давайте предположим, что у этого приложения есть две основные задачи: возможность загружать (записывать) изображения на сервер и запрашивать изображения. Безусловно, эффективная загрузка является важным критерием, однако приоритетом будет быстрая доставка по запросу пользователей (например, изображения могут быть запрошены для отображения на веб-странице или другим приложением). Эта функциональность аналогична той, которую может обеспечить веб-сервер или граничный сервер Сети доставки

контента (Content Delivery Network, CDN). Сервер CDN обычно хранит объекты данных во многих расположениях, таким образом, их географическое/физическое размещение оказывается ближе к пользователям, что приводит к росту производительности.

Другие важные аспекты системы:

- Количество хранимых изображений может быть безгранично, таким образом, масштабируемость хранения необходимо рассматривать именно с этой точки зрения.
- Должна быть низкая задержка для загрузок/запросов изображения.
- Если пользователь загружает изображение на сервер, то его данные должны всегда оставаться целостными и доступными.
- Система должна быть простой в обслуживании (управляемость).
- Так как хостинг изображений не приносит большой прибыли, система должна быть экономически эффективной.

Рисунок 1.1 представляет собой упрощенную схему функциональности.

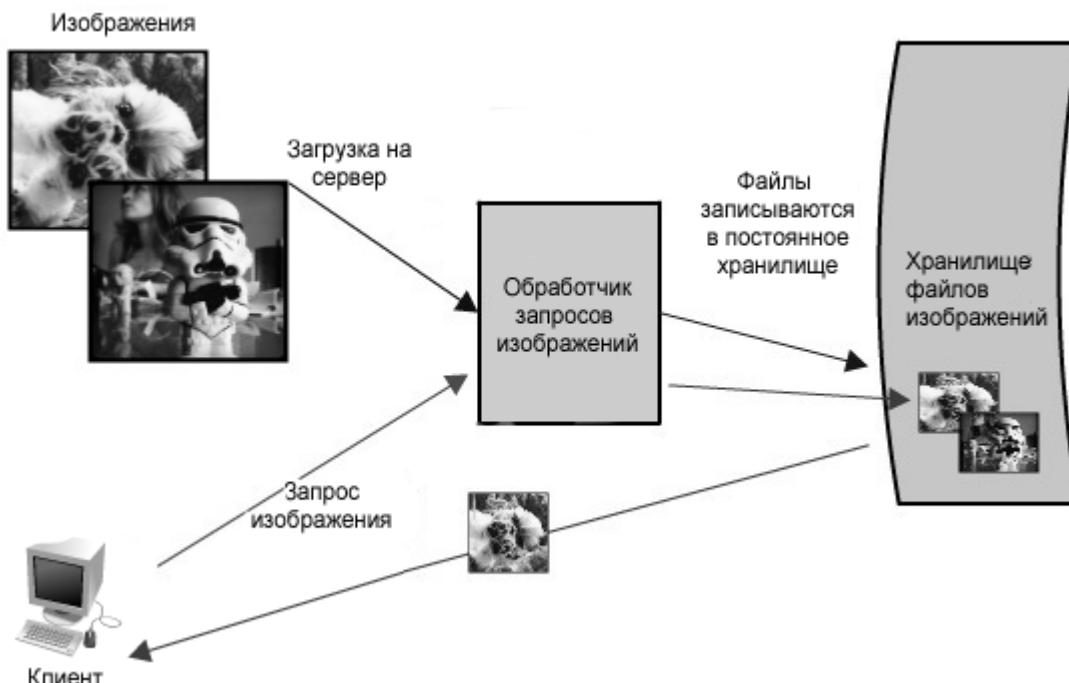


Рисунок 1.1: Упрощенная схема архитектуры для приложения хостинга изображений

В этом примере хостинга изображений система должна быть заметно быстрой, ее данные надежно сохранены, и все эти атрибуты хорошо масштабирумы. Создание небольшой версии этого приложения представляло бы из себя стандартную задачу, и для его размещения достаточно было бы единственного сервера. Однако, подобная ситуация не представляла бы интереса для данной главы. Давайте предположим, что нам необходимо создать что-то такое же масштабное, как Flickr.

Сервисы

При рассмотрении дизайна масштабируемой системы, бывает полезным разделить функциональность и подумать о каждой части системы как об отдельной службе с четко определенным интерфейсом. На практике считается, что системы разработанные таким образом имеют Service-Oriented Architecture (SOA). Для этих типов систем у каждой службы существует свой собственный отличный функциональный контекст, и взаимодействие с чем-либо за пределами этого контекста происходит через абстрактный интерфейс, обычно общедоступный API другой службы.

Деконструкция системы на ряд комплементарных сервисов изолирует работу одних частей от дру-

гих. Эта абстракция помогает устанавливать четкие отношения между службой, ее базовой средой и потребителями службы. Создание четкой схемы может помочь локализовать проблемы, но также и позволяет каждой части масштабироваться независимо друг от друга. Этот вид сервисно-ориентированного дизайна систем для обслуживания широкого круга запросов аналогичен объектно-ориентированному подходу в программировании.

В нашем примере все запросы загрузки и получения изображения обработаны одним и тем же сервером; однако, поскольку система должна масштабироваться, целесообразно выделить эти две функции в их собственные сервисы.

Предположим, что в будущем служба находится в интенсивном использовании; такой сценарий помогает лучше проследить, как более длительные записи влияют на время для считывания изображения (так как две функции будут конкурировать за совместно используемые ресурсы). В зависимости от архитектуры этот эффект может быть существенным. Даже если скорость отдачи и приема будут одинаковы (что не характерно для большинства сетей IP, поскольку они разработаны для соотношения скорости приема к скорости отдачи как минимум 3:1), считываемые файлы будут обычно извлекаться из кэша, и записи должны, в конечном счете, попасть на диск (и возможно подвергнуться многократной перезаписи в похожих ситуациях). Даже если все данные будут в памяти или читаться с дисков (таких как твердотельные диски SSD), то записи в базу данных почти всегда будут медленнее, чем чтения из нее. (Pole Position, инструмент с открытым исходным кодом для сравнительного тестирования баз данных, <http://polepos.org/> и результаты <http://polepos.sourceforge.net/results/PolePositionClientServer.pdf>).

Другая потенциальная проблема с этим дизайном состоит в том, что у веб-сервера, такого как Apache или lighttpd обычно существует верхний предел количества одновременных соединений, которые он в состоянии обслужить (значение по умолчанию - приблизительно 500, но оно может быть намного выше), и при высоком трафике записи могут быстро израсходовать этот предел. Так как чтения могут быть асинхронными или использовать в своих интересах другую оптимизацию производительности как gzip-сжатие или передача с делением на порции, веб-сервер может переключить чтения подачи быстрее и переключиться между клиентами, обслуживая гораздо больше запросов, чем максимальное число соединений (с Apache и максимальным количеством соединений, установленном в 500, вполне реально обслуживать несколько тысяч запросов чтения в секунду). Записи, с другой стороны, имеют тенденцию поддерживать открытое соединение на протяжении всего времени загрузки. Так передача файла размером 1 МБ на сервер могла занять больше 1 секунды в большинстве домашних сетей, в результате веб-сервер сможет обработать только 500 таких одновременных записей.

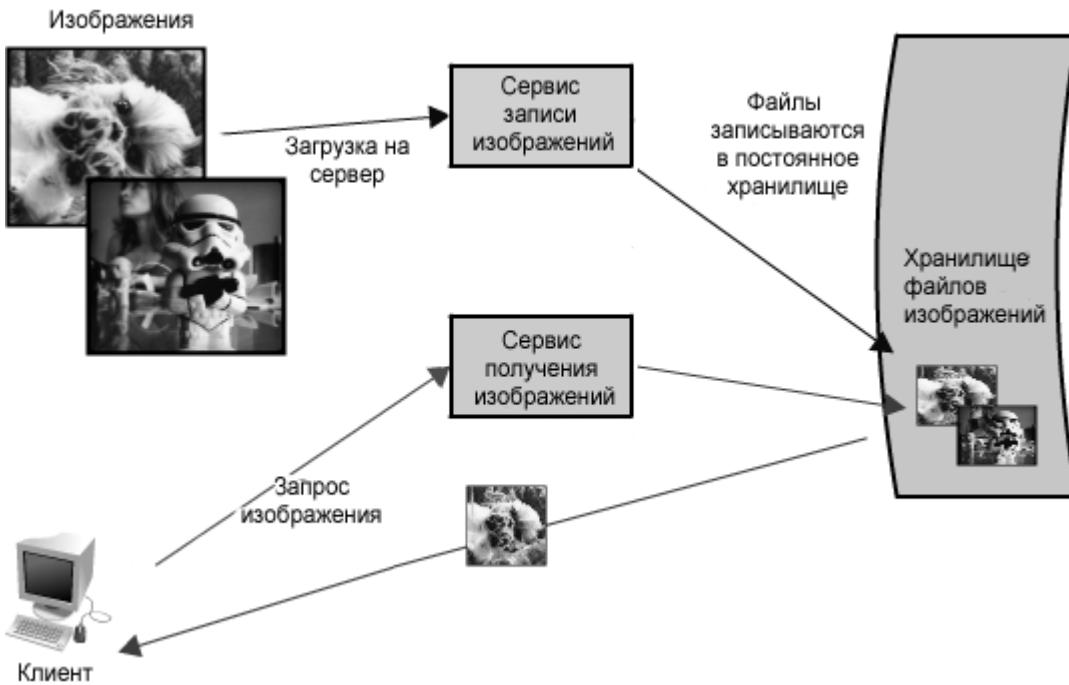


Рисунок 1.2: Разделение чтения и записи

Предвидение подобной потенциальной проблемы свидетельствует о необходимости разделения чтения и записи изображений в независимые службы, показанные на рисунке 1.2. Это позволит не только масштабировать каждую из них по отдельности (так как вероятно, что мы будем всегда делать больше чтений, чем записей), но и быть в курсе того, что происходит в каждой службе. Наконец, это разграничит проблемы способные возникнуть в будущем, что упростит диагностику и оценку проблемы медленного доступа на чтение.

Преимущество этого подхода состоит в том, что мы в состоянии решить проблемы независимо друг от друга - при этом нам не придется думать о необходимости записи и получении новых изображений в одном контексте. Обе из этих служб все еще используют глобальный корпус изображений, но при использовании методов соответствующих определенной службе, они способны оптимизировать свою собственную производительность (например, помещая запросы в очередь, или кэшируя популярные изображения - более подробно об этом речь пойдет далее). Как с точки зрения обслуживания, так и стоимости каждая служба может быть масштабирована независимо по мере необходимости. И это является положительным фактором, поскольку их объединение и смешивание могло бы непреднамеренно влиять на их производительность, как в сценарии, описанном выше.

Конечно, работа вышеупомянутой модели будет оптимальной, в случае наличия двух различных конечных точек (фактически, это очень похоже на несколько реализаций провайдеров "облачного" хранилища и Сетей доставки контента). Существует много способов решения подобных проблем, и в каждом случае можно найти компромисс.

К примеру, Flickr решает эту проблему чтения-записи, распределяя пользователи между разными модулями, таким образом, что каждый модуль может обслуживать только ограниченное число определенных пользователей, и когда количество пользователи увеличиваются, больше модулей добавляется к кластеру (см. презентацию масштабирования Flickr, <http://mysqldb.blogspot.com/2008/04/mysql-uc-2007-presentation-file.html>). В первом примере проще масштабировать аппаратные средства на основе фактической нагрузки использования (число чтений и записей во всей системе), тогда как масштабирование Flickr происходит на основе базы пользователей(однако, здесь используется предположение равномерного использования у разных пользователей, таким образом, мощность нужно планировать с запасом). В прошлом недоступность или проблема с одной из служб приводили в нерабочее состояние функциональность целой систе-

мы (например, никто не может записать файлы), тогда недоступность одного из модулей Flickr будет влиять только на пользователей, относящихся к нему. В первом примере проще выполнить операции с целым набором данных - например, обновляя службу записи, чтобы включить новые метаданные, или выполняя поиск по всем метаданным изображений - тогда как с архитектурой Flickr каждый модуль должен был быть подвергнут обновлению или поиску (или поисковая служба должна быть создана, чтобы сортировать те метаданные, которые фактически для этого и предназначены).

Что касается этих систем - не существует никакой панацеи, но всегда следует исходить из принципов, описанных в начале этой главы: определить системные потребности (нагрузка операциями "чтения" или "записи" или всем сразу, уровень параллелизма, запросы по наборам данных, диапазоны, сортировки, и т.д.), провести сравнительное эталонное тестирование различных альтернатив, понять условия потенциального сбоя системы и разработать комплексный план на случай возникновения отказа.

Избыточность

Чтобы элегантно справится с отказом, у веб-архитектуры должна быть избыточность ее служб и данных. Например, в случае наличия лишь одной копии файла, хранившегося на единственном сервере, потеря этого сервера будет означать потерю и файла. Вряд ли подобную ситуацию можно положительно охарактеризовать, и обычно ее можно избежать путем создания множественных или резервных копий.

Этот тот же принцип применим и к службам. От отказа единственного узла можно защититься, если предусмотреть неотъемлемую часть функциональности для приложения, гарантирующую одновременную работу его нескольких копий или версий.

Создание избыточности в системе позволяет избавиться от слабых мест и обеспечить резервную или избыточную функциональность на случай непредвиденной ситуации. Например, в случае наличия двух экземпляров одной и той же службы, работающей в "продакшн", и один из них выходит из строя полностью или частично, система может преодолеть отказ за счет *переключения на исправный экземпляр*.

Переключение может происходить автоматически или потребовать ручного вмешательства..

Другая ключевая роль избыточности службы - создание *архитектуры, не предусматривающей разделения ресурсов*. С этой архитектурой каждый узел в состоянии работать самостоятельно и, более того, в отсутствие центрального , управляющего состояниями или координирующего действия других узлов. Она способствует масштабируемости, так как добавление новых узлов не требует специальных условий или знаний. И что наиболее важно, в этих системах не найдется никакой критически уязвимой точки отказа, что делает их намного более эластичными к отказу..

Например, в нашем приложении сервера изображения, все изображения имели бы избыточные копии где-нибудь в другой части аппаратных средств (идеально - с различным географическим местоположением в случае такой катастрофы, как землетрясение или пожар в центре обработки данных), и службы получения доступа к изображениям будут избыточны, при том, что все они потенциально будут обслуживать запросы. (См. рисунок 1.3.)

Забегая вперед, балансирующие нагрузки - отличный способ сделать это возможным, но подробнее об этом ниже.

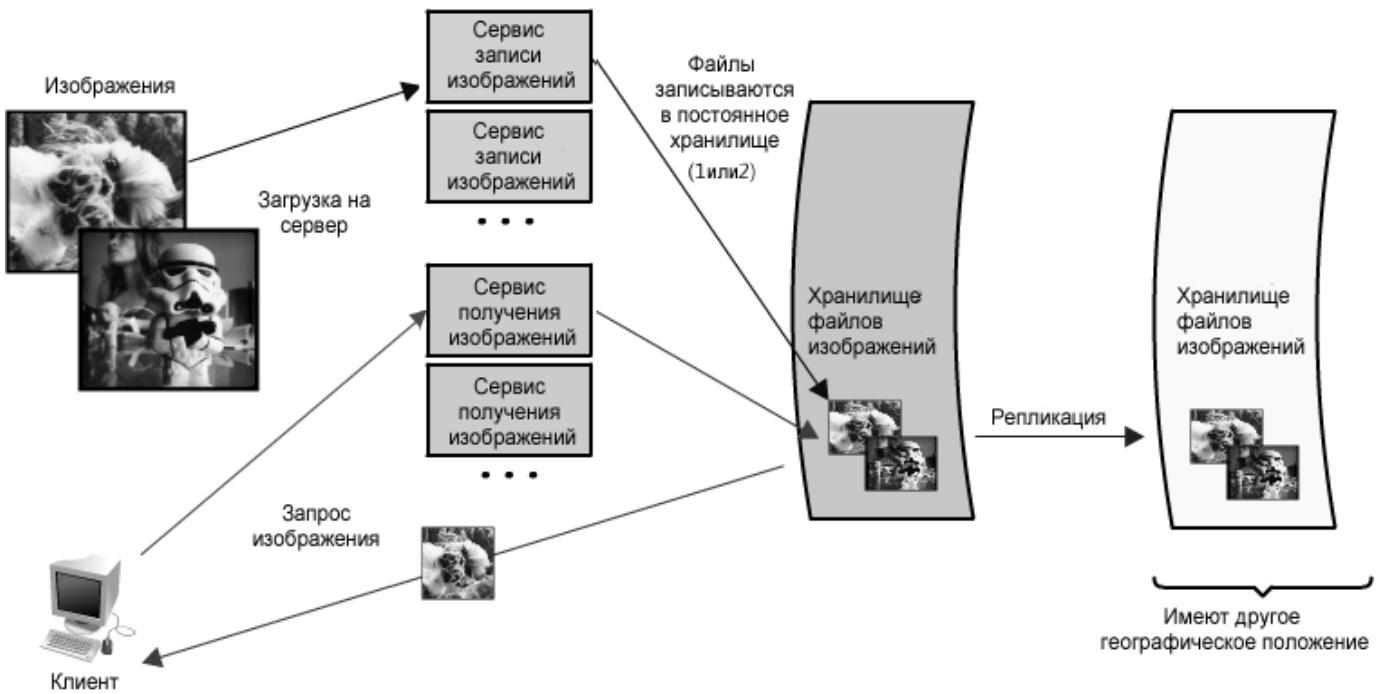


Рисунок 1.3: Приложение хостинга изображений с избыточностью

Сегментирование

Наборы данных могут быть настолько большими, что их невозможно будет разместить на одном сервере. Может также случиться, что вычислительные операции потребуют слишком больших компьютерных ресурсов, уменьшая производительность и делая необходимым увеличение мощности. В любом случае у вас есть два варианта: вертикальное или горизонтальное масштабирование.

Вертикальное масштабирование предполагает добавление большего количества ресурсов к отдельному серверу. Так, для очень большого набора данных это означало бы добавление большего количества (или большего объема) жестких дисков, и таким образом весь набор данных мог бы разместиться на одном сервере. В случае вычислительных операций это означало бы перемещение вычислений в более крупный сервер с более быстрым ЦП или большим количеством памяти. В любом случае, вертикальное масштабирование выполняется для того, чтобы сделать отдельный ресурс вычислительной системы способным к дополнительной обработке данных.

Горизонтальное масштабирование, с другой стороны, предполагает добавление большего количества узлов. В случае большого набора данных это означало бы добавление второго сервера для хранения части всего объема данных, а для вычислительного ресурса это означало бы разделение работы или загрузки через некоторые дополнительные узлы. Чтобы в полной мере воспользоваться потенциалом горизонтального масштабирования, его необходимо реализовать как внутренний принцип разработки архитектуры системы. В противном случае изменение и выделение контекста, необходимого для горизонтального масштабирования может оказаться проблематичным.

Наиболее распространенным методом горизонтального масштабирования считается разделение служб на сегменты или модули. Их можно распределить таким образом, что каждый логический набор функциональности будет работать отдельно. Это можно сделать по географическими границами, или другим критериям таким, как платящие и не платящие пользователи. Преимущество этих схем состоит в том, что они предоставляют услугу или хранилище данных с расширенной функциональностью.

В нашем примере сервера изображения, возможно, что единственный файловый сервер, используемый для хранения изображения, можно заменить множеством файловых серверов, при этом

каждый из них будет содержать свой собственный уникальный набор изображений. (См. рисунок 1.4.) Такая архитектура позволит системе заполнять каждый файловый сервер изображениями, добавляя дополнительные серверы, по мере заполнения дискового пространства. Дизайн потребует схемы именования, которая свяжет имя файла изображения с содержащим его сервером. Имя изображения может быть сформировано из консистентной схемы хеширования, привязанной к серверам. Или альтернативно, каждое изображение может иметь инкрементный идентификатор, что позволит службе доставки при запросе изображения обработать только диапазон идентификаторов, привязанных к каждому серверу (в качестве индекса).

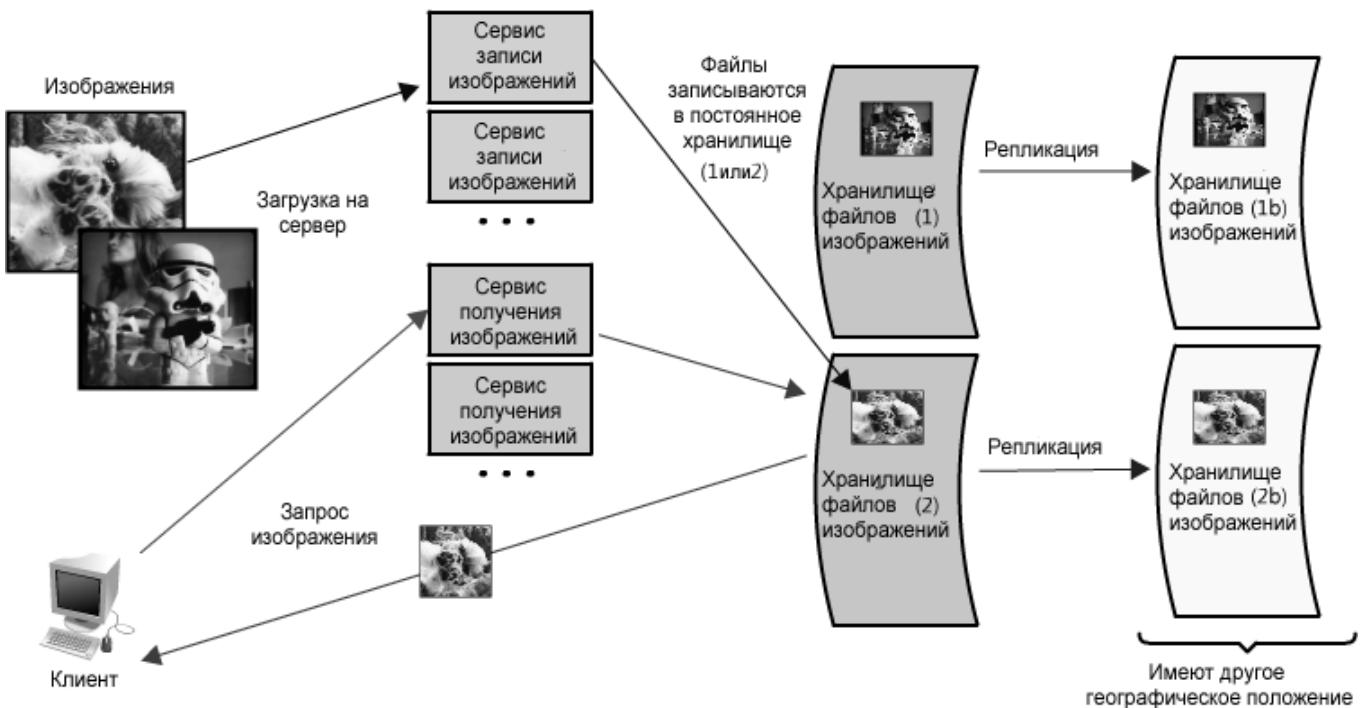


Рисунок 1.4: Приложение хостинга изображений с избыточностью и сегментированием

Конечно, есть трудности в распределении данных или функциональности на множество серверов. Один из ключевых вопросов - *местоположение данных*; в распределенных системах, чем ближе данные к месту проведения операций или точке вычисления, тем лучше производительность системы. Следовательно, распределение данных на множество серверов потенциально проблематично, так как в любой момент, когда эти данные могут понадобиться, появляется риск того, что их может не оказаться по месту требования, серверу придется выполнить затратную выборку необходимой информации по сети.

Другая потенциальная проблема возникает в форме *несогласованности (неконсистентности)*. Когда различные сервисы выполняют считывание и запись на совместно используемом ресурсе, потенциально другой службе или хранилище данных, существует возможность возникновения условий "состязания" - где некоторые данные считаются обновленными до актуального состояния, но в реальности их считывание происходит до момента актуализации - и таком случае данные неконсистентны. Например, в сценарии хостинга изображений, состояние состязания могло бы возникнуть в случае, если бы один клиент отправил запрос обновления изображения собаки с изменением заголовка "Собака" на "Гизмо", в тот момент, когда другой клиент считывал изображение. В такой ситуации неясно, какой именно заголовок, "Собака" или "Гизмо", был бы получен вторым клиентом..

Есть, конечно, некоторые препятствия, связанные с сегментированием данных, но сегментирование позволяет выделять каждую из проблем из других: по данным, по загрузке, по образцам использования, и т.д. в управляемые блоки. Это может помочь с масштабируемостью и управляемостью, но риск все равно присутствует. Есть много способов уменьшения риска и обработки сбоев; однако, в интересах краткости они не охвачены в этой главе. Если Вы хотите получить больше

информации по данной теме, вам следует взглянуть на [блог-пост](#) по отказоустойчивости и мониторингу.

1.3. Структурные компоненты быстрого и масштабируемого доступа к данным

Рассмотрев некоторые базовые принципы в разработке распределенных систем, давайте теперь перейдем к более сложному моменту - масштабирование доступа к данным.

Самые простые веб-приложения, например, приложения стека LAMP, схожи с изображением на рисунке 1.5.



Рисунок 1.5: Простые веб-приложения

С ростом приложения возникают две основных сложности: масштабирование доступа к серверу приложений и к базе данных. В хорошо масштабируемом дизайне приложений веб-сервер или сервер приложений обычно минимизируется и часто воплощает архитектуру, не предусматривающую совместного разделения ресурсов. Это делает уровень сервера приложений системы горизонтально масштабируемым. В результате использования такого дизайна тяжёлый труд смещается вниз по стеку к серверу базы данных и вспомогательным службам; именно на этом слое и вступают в игру настоящие проблемы масштабирования и производительности.

Остальная часть этой главы посвящена некоторым наиболее распространенным стратегиям и методам повышения производительности и обеспечения масштабируемости подобных типов служб путем предоставления быстрого доступа к данным.

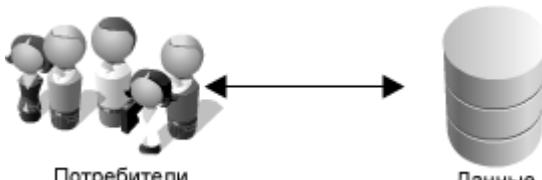


Рисунок 1.6: Упрощенное веб-приложение

Большинство систем может быть упрощено до схемы на рисунке 1.6, которая является хорошей отправной точкой для начала рассмотрения. Если у Вас есть много данных, можно предположить, что Вы хотите иметь к ним такой же легкий доступ и быстрый доступ, как к коробке с леденцами в верхнем ящике вашего стола. Хотя данное сравнение чрезмерно упрощено, оно указывает на две сложные проблемы: масштабируемость хранилища данных и быстрый доступ к данным.

Для рассмотрения данного раздела давайте предположим, что у Вас есть много терабайт (ТБ) данных, и Вы позволяете пользователям получать доступ к небольшим частям этих данных в произвольном порядке. (См. рисунок 1.7.)

Схожей задачей является определение местоположения файла изображения где-нибудь на файло-

вом сервере в примере приложения хостинга изображений.

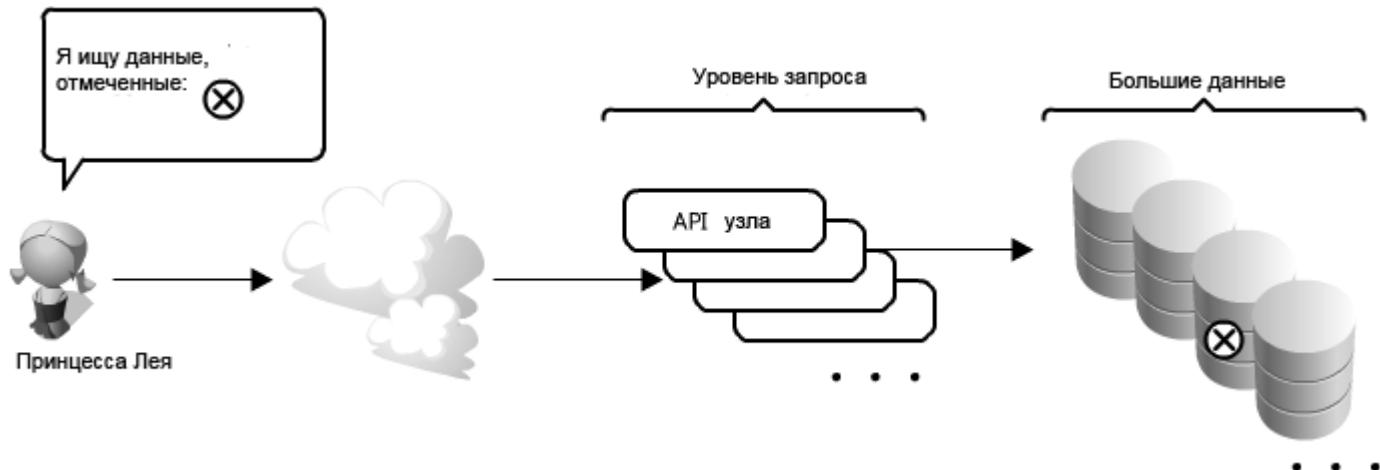


Рисунок 1.7: Доступ к определенным данным

Это особенно трудно, потому что загрузка терабайтов данных в память может быть очень накладной и непосредственно влияет на количество дисковых операций ввода-вывода. Скорость чтения с диска в несколько раз ниже скорости чтения из оперативной памяти - можно сказать, что доступ к памяти с так же быстр, как Чак Норрис, тогда как доступ к диску медленнее очереди в поликлинике. Эта разность в скорости особенно ощутима для больших наборов данных; в сухих цифрах доступ к памяти 6 раз быстрее, чем чтение с диска для последовательных операций чтения, и в 100,000 раз - для чтений в случайному порядке (см. "Патологии Больших Данных", <http://queue.acm.org/detail.cfm?id=1563874>). Кроме того, даже с уникальными идентификаторами, решение проблемы нахождения местонахождения небольшой порции данных может быть такой же трудной задачей, как и попытка не глядя вытащить последнюю конфету с шоколадной начинкой из коробки с сотней других конфет.

К счастью существует много подходов, которые можно применить для упрощения, из них четыре наиболее важных подхода - это использование кэшей, прокси, индексов и балансировщиков нагрузки. В оставшейся части этого раздела обсуждается то, как каждое из этих понятий может быть использовано для того, чтобы сделать доступ к данным намного быстрее.

Кэши

Кэширование дает выгоду за счет характерной черты базового принципа: недавно запрошенные данные вполне вероятно потребуются еще раз. Кэши используются почти на каждом уровне вычислений: аппаратные средства, операционные системы, веб-браузеры, веб-приложения и не только. Кэш походит на кратковременную память: ограниченный по объему, но более быстрый, чем исходный источник данных, и содержащий элементы, к которым недавно получали доступ. Кэши могут существовать на всех уровнях в архитектуре, но часто находятся на самом близком уровне к фронтэнду, где они реализованы, чтобы возвратить данные быстро без значительной нагрузки бэкэнда.

Каким же образом кэш может использоваться для ускорения доступа к данным в рамках нашего примера API? В этом случае существует несколько мест, подходящих размещения кэша. В качестве одного из возможных вариантов размещения можно выбрать узлы на уровне запроса, как показано на рисунке 1.8.

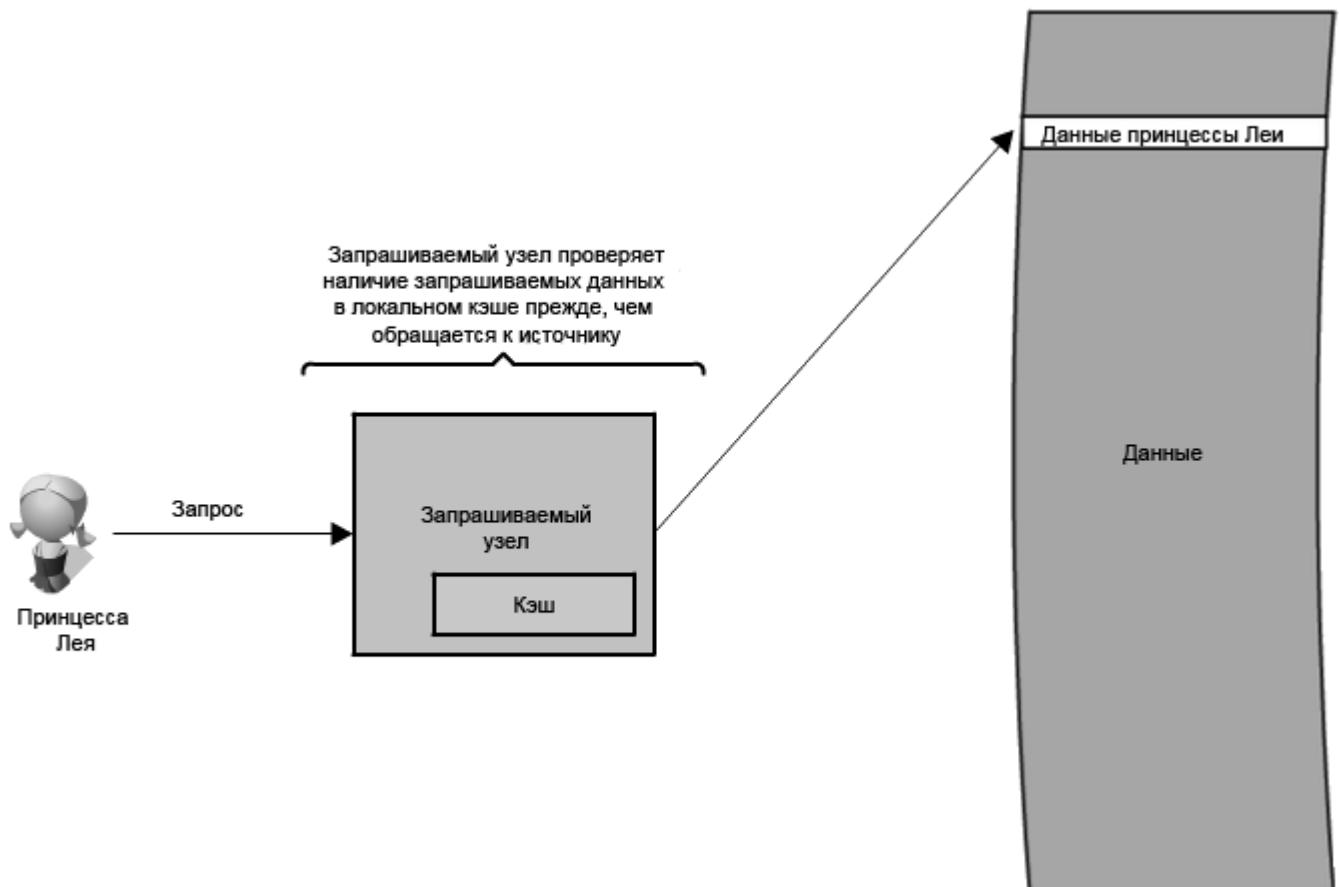


Рисунок 1.8: Размещение кэша на узле уровня запроса

Размещение кэша непосредственно на узле уровня запроса позволяет локальное хранение данных ответа. Каждый раз, когда будет выполняться запрос к службе, узел быстро возвратит локальные, кэшированные данные, если таковые существуют. Если это не будет в кэше, то узел запроса запросит данные от диска. Кэш на одном узле уровня запроса мог также быть расположен как в памяти (которая очень быстра), так и на локальном диске узла (быстрее, чем попытка обращения к сетевому хранилищу).

Каждый запрашиваемый узел проверит
локальный кэш перед обращением
к источнику

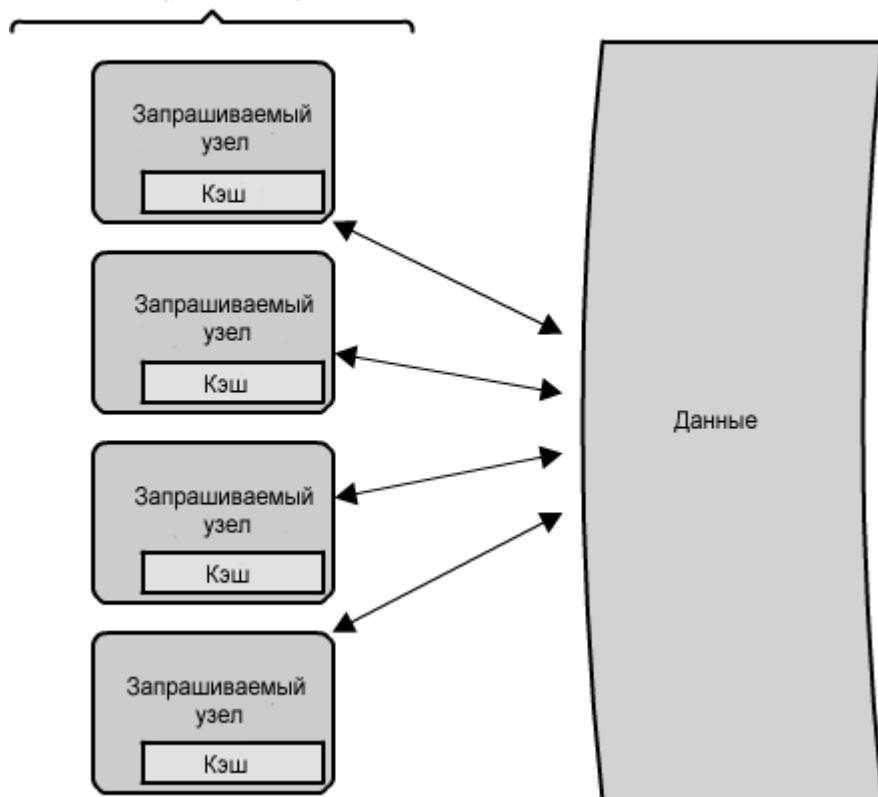


Рисунок 1.9: Системы кэшей

Что происходит, когда вы распространяете кеширование на множество узлов? Как Вы видите в рисунке 1.9, если уровень запроса будет включать множество узлов, то вполне вероятно, что каждый узел будет и свой собственный кэш. Однако, если ваш балансировщик нагрузки в произвольном порядке распределит запросы между узлами, то тот же запрос перейдет к различным узлам, таким образом увеличивая неудачные обращения в кэш. Двумя способами преодоления этого препятствия являются глобальные и распределенные кэши.

Глобальный кэш

Смысл глобального кэша понятен из названия: все узлы используют одно единственное пространство кэша. В этом случае добавляется сервер или хранилище файлов некоторого вида, которые быстрее, чем Ваше исходное хранилище и, которые будут доступны для всех узлов уровня запроса. Каждый из узлов запроса запрашивает кэш таким же образом, как если бы он был локальным. Этот вид кэширующей схемы может вызвать некоторые затруднения, так как единственный кэш очень легко перегрузить, если число клиентов и запросов будет увеличиваться. В тоже время такая схема очень эффективна при определенной архитектуре (особенно связанной со специализированными аппаратными средствами, которые делают этот глобальный кэш очень быстрым, или у которых есть фиксированный набор данных, который должен кэшироваться).

Есть две стандартных формы глобальных кэшей, изображенных в схемах. На рисунке 1.10 изображена ситуация, когда кэшируемый ответ не найден в кэше, сам кэш становится ответственным за получение недостающей части данных от базового хранилища. На рисунке 1.11 проиллюстрирована обязанность узлов запроса получить любые данные, которые не найдены в кэше.

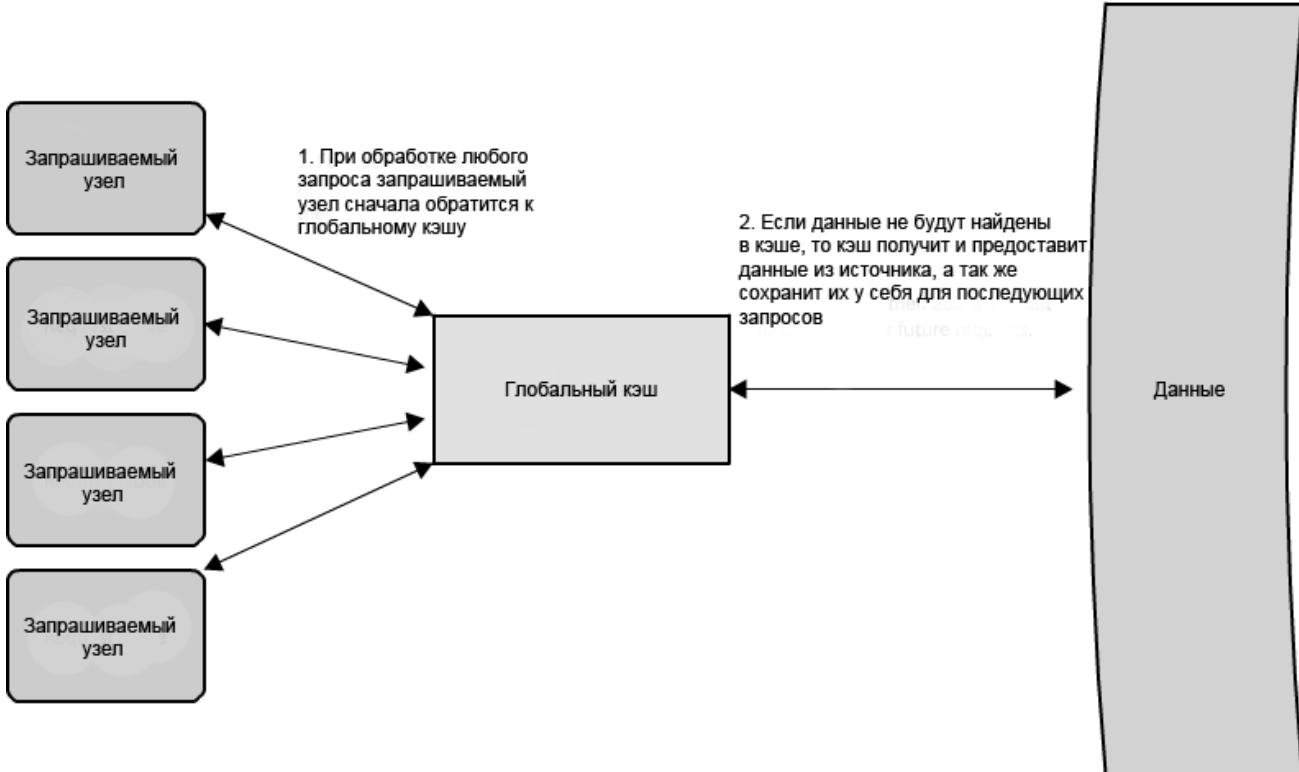


Рисунок 1.10: Глобальный кэш, где кэш ответственен за извлечение

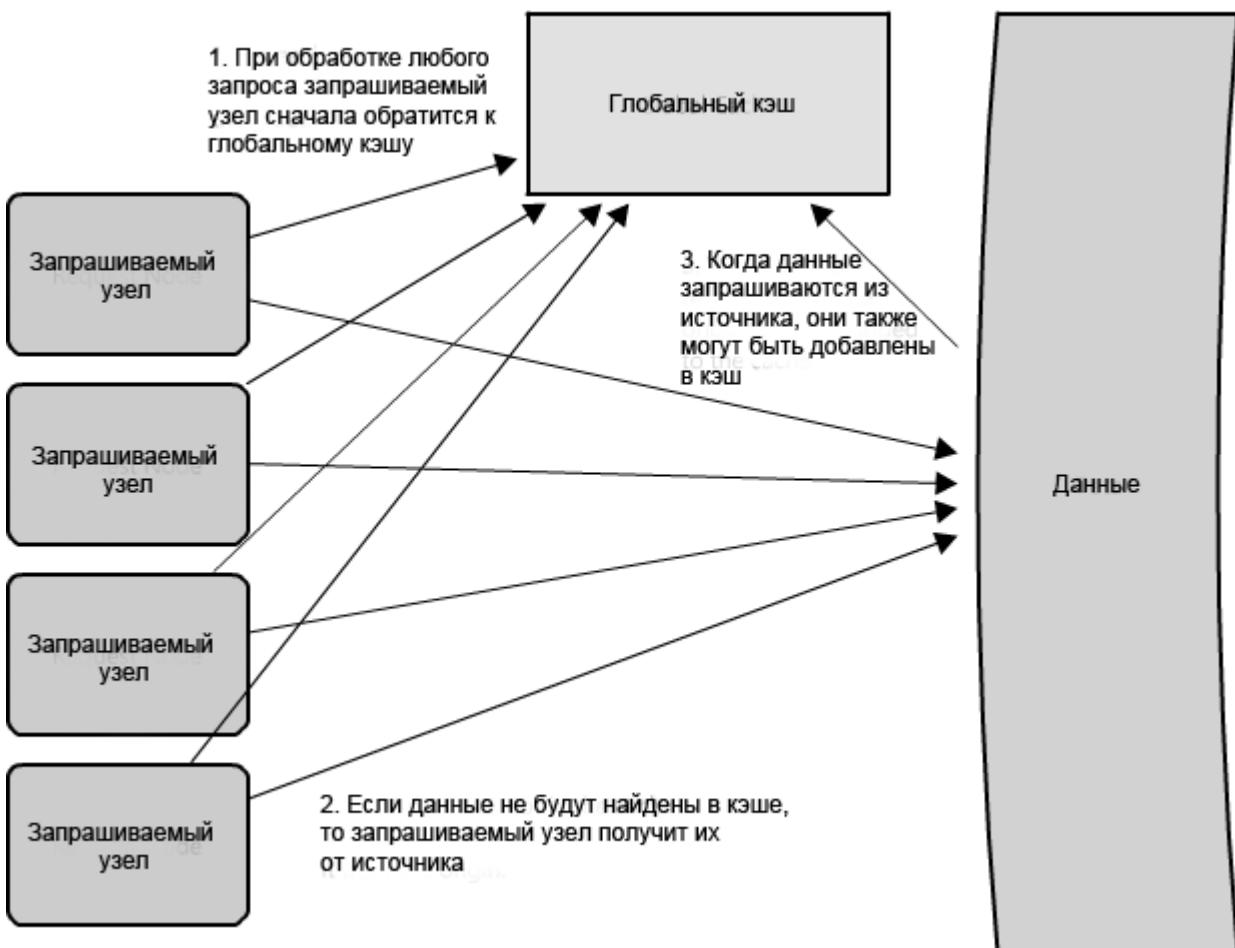


Рисунок 1.11: Глобальный кэш, где узлы запроса ответственны за извлечение

Большинство приложений, усиливающих глобальные кэши, склонно использовать первый тип, где сам кэш управляет замещением и данными выборки, чтобы предотвратить лавинную рассылку запросов на те же данные от клиентов. Однако, есть некоторые случаи, где вторая реализация имеет больше смысла. Например, если кэш используется для очень больших файлов, низкий процент

удачного обращения в кэш приведет к перегрузке кэша буфера неудачными обращениями в кэш; в этой ситуации это помогает иметь большой процент общего набора данных (или горячего набора данных) в кэше. Другой пример - архитектура, где файлы, хранящиеся в кэше, статичны и не должны быть удалены. (Это может произойти из-за основных эксплуатационных характеристик касательно такой задержки данных - возможно, определенные части данных должны оказаться очень быстрыми для больших наборов данных - когда логика приложения понимает стратегию замещения или горячие точки лучше, чем кэш.)

Распределенный кэш

В распределенном кэше (рисунок 1.12), каждый из его узлов владеет частью кэшированных данных, поэтому если холодильник в продуктовом магазине сравнить с кэшем, тогда распределенный кэш походит на хранение вашей еды в нескольких удобных для доступа местах - холодильнике, стойках и коробке для завтрака, что избавляет необходимости совершать путешествия на склад. Обычно кэш сегментирован при помощи непротиворечивой хеш-функции. Если узел запроса ищет определенную часть данных, он может быстро узнать, куда смотреть в распределенном кэше, чтобы определить, доступны ли эти данные. В этом случае каждый узел поддерживает маленькую часть кэша и сначала отправляет запрос данных другому узлу прежде, чем обращается к источнику. Поэтому, одно из преимуществ распределенного кэша - расширяемое пространство кэша, что достигается простым добавлением узлов к пулу обработки запросов.

Недостаток распределенного кэширования - работа в условиях недостающих узлов. Некоторые распределенные кэши обходят эту проблему, храня избыточные копии данных на множестве узлов; однако, можно представить, насколько быстро логическая структура такого кэша может сложной, особенно в условиях добавления или удаления узлов из уровня запроса. Стоит отметить - даже если узел исчезает, и часть кэша будет потеряна, последствия неизбежно окажутся катастрофическими - запросы просто получат данные непосредственно от источника!

При обработке любого запроса запрашиваемый узел обращается кешу по имени ключа (используя предопределенный алгоритм консистентного хеширования) соответствующего элемента данных и в случае отсутствия - к источнику данных

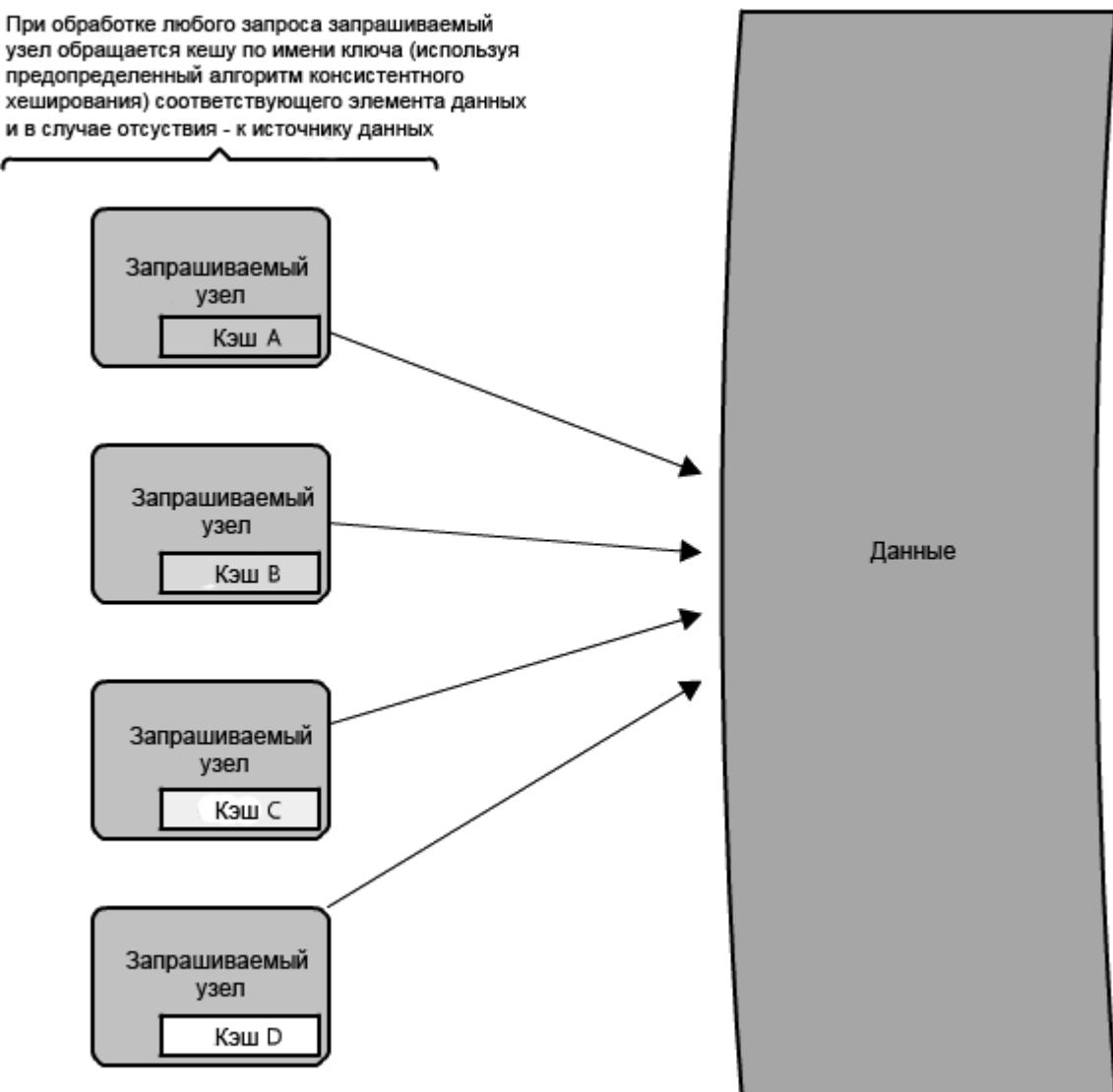


Рисунок 1.12: Распределенный кэш.

Большим преимуществом кэшей является увеличение скорости работы системы (безусловно, только при правильной реализации!) Выбранная методология позволяет ускорить этот процесс для еще большего количества запросов. Однако, использование кэширования предполагает определенные затраты на поддержание дополнительного пространства обычно дорогостоящей памяти. Кэши замечательно подходят не только для общего увеличения производительности системы, но и обеспечения ее функциональность при нагрузке такого высокого уровня, которая в обычной ситуации привела бы к полному отказу в обслуживании.

Одним из популярных примеров кэша с открытым исходным кодом можно назвать Memcached (), который может работать как локальным, так и распределенным кэшем); кроме того, есть много других вариантов (включая специфичные для определенного языка или платформы).

Memcached используется во многих больших веб-сайтах, и даже при том, что он может быть очень мощным, представляет собой просто хранилище типа ключ-значение в оперативной памяти, оптимизированного для произвольного хранения данных и быстрых поисков ($O(1)$).

Facebook использует несколько различных типов кэширования, чтобы добиться высокой производительности своего сайта (см., ["Facebook: кэширование и производительность"](#)). Они используют \$GLOBALS и APC, кэширующие на уровне языка (представленные в PHP за счет вызова функции), который способствует ускорению промежуточных вызовов функции и получению результатов.

(Большинство языков оснащены этими типами библиотек для улучшения производительность веб-страницы, и они почти всегда должны использоваться.) Кроме того Facebook использует глобальный кэш, который распределен на множество серверов (см. "[Масштабирование memcached в Facebook](#)"), таким образом, что один вызов функции, получающий доступ к кэшу, мог параллельно выполнить множество запросов для данных, хранящихся на различных серверах Memcached. Такой подход позволяет добиться намного более высокой производительности и пропускной способности для данных профиля пользователя, и создать централизованную архитектуру обновления данных. Это важно, так как, при наличии тысяч серверов, функции аннулирования и поддержания непротиворечивости кэша могут вызывать затруднение.

Далее речь пойдет об алгоритме действий в случае отсутствия данных в кэше.

Прокси

На базовом уровне прокси-сервер - промежуточная часть аппаратных средств/программного обеспечения, которые получают запросы от клиентов и передают их к серверам источника бэкэнда. Как правило, прокси используются, чтобы фильтровать запросы, протоколировать запросы, или иногда преобразовывать запросы (добавляя/удаляя заголовки, шифруя/десифруя или сжимая).



Рисунок 1.13 Прокси-сервер

Прокси также очень полезны при координировании запросов, поступающих от большого количества серверов, что дает возможность оптимизировать трафик запроса в масштабе всей системы. Один из способов использования прокси для ускорения доступа к данным заключается в объединении одинаковых или схожих запросов и передачи единого ответа клиентам запроса. Этот термин получил название сжатое перенаправление (collapsed forwarding).

Представим, что с нескольких узлов поступают запросы на одинаковые данные (назовем их littleB), но в кэше часть этих данных отсутствует. Если этот запрос направляется через прокси, то все запросы могут быть объединены в один, и в результате этой оптимизации littleB будет считан с диска только один раз. (См. рисунок 1.14) В этом случае придется немного пожертвовать скоростью, поскольку процесс обработки запросов и их объединения привести к несколько более длительным задержкам. Однако, при высокой нагрузке это напротив приведет к улучшению производительности, особенно в случае многократных запросов одинаковых данных. Стратегия функционирования прокси аналогична кэшу, но вместо хранения данных, он оптимизирует запросы или вызовы документам.

В LAN-прокси, например, клиенты не нуждаются в своем собственном IP-адресе, для соединения с Интернетом. Прокси объединяет запросы от клиентов на одинаковый контент. Однако это порождает двусмысленность, так как многие прокси являются также и кэшами (поскольку являются логичным местом для размещения кэша), но не все кэши работают как прокси.

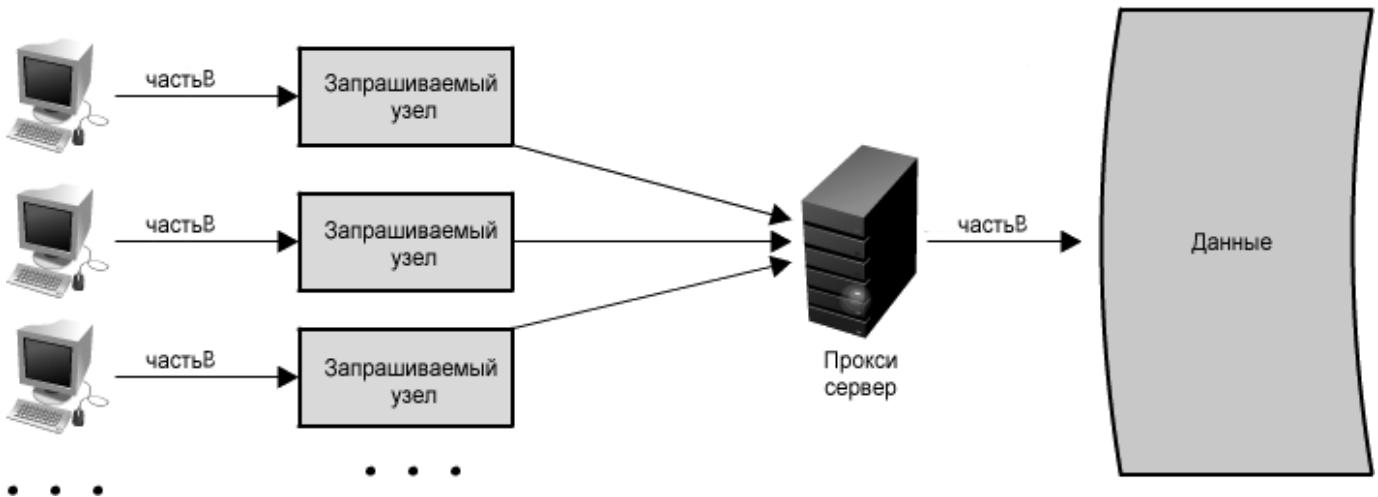


Рисунок 1.14: Использование прокси-сервера для комбинирования запросов

Еще одним отличным способом использования прокси состоит не просто в объединении запросов одинаковых данных, но также и частей данных, которые находятся пространственно близко друг к другу в хранилище источника (последовательно на диске). Использование такой стратегии максимизирует локальность данных для запросов, что может привести к сокращению задержки запроса. Например, если набор узлов запрашивает части В: часть-В1, часть-В2, и т.д., мы можем настроить наш прокси, чтобы он распознавал пространственное местоположение отдельных запросов, комбинируя их в единственный запрос и возвращаясь только bigB, значительно минимизируя чтения из источника данных. (См. рисунок 1.15) В случае доступа к целым терабайтам данных в произвольном порядке время реализации запроса может сильно отличаться. Так как прокси могут, по существу, сгруппировать несколько запросов в один, они особенно полезны в ситуациях с высокой нагрузкой или ограниченными возможностями кэширования.

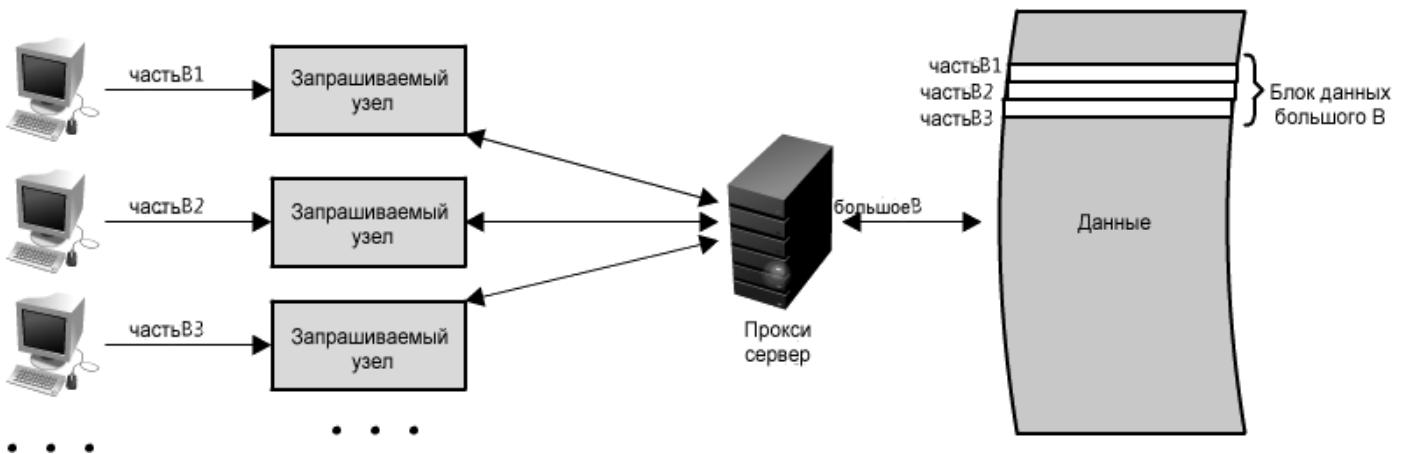


Рисунок 1.15: Использование прокси для комбинирования запросов на данные, находящихся пространственно близко друг к другу

Стоит отметить, что вы можете использовать прокси и кэши вместе, но обычно лучше помещать кэш перед прокси по той же причине, по которой лучше позволять более быстрым бегунам стартовать в марафоне с большим количеством участников. Это вызвано тем, что кэш использует данные из памяти, что очень быстро, и это не противоречит многократным запросам на тот же результат. Но если бы кэш был расположен с другой стороны прокси-сервера, то возникла бы дополнительная задержка для каждого запроса перед кэшем, что могло бы снизить производительность.

Если вы рассматриваете добавление прокси в ваши системы, то у вас есть много вариантов для

выбора;

[Squid](#) и

[Varnish](#) прошли испытания временем и широко используются во многих производительных веб-сайтах. Эти решения для прокси предлагают множество вариантов оптимизации, чтобы выжать максимум из клиент-серверного обмена данными. Установка одного из них в режиме реверсивного прокси (описан ниже в разделе о балансирующей нагрузке) на уровне веб-сервера может значительно улучшить производительность веб-сервера, уменьшая объем работы для обработки входящих клиентских запросов.

Индексы

Использование индекса для получения быстрого доступа к вашим данным - известная стратегия для того, чтобы эффективно оптимизировать доступ к данным. Наиболее широкое применение индексирования находит в базах данных. Индекс делает взаимные уступки, используя издержки объемов хранения данных и снижая скорости операций (так как вы должны одновременно и записывать данные, и обновлять индекс), позволяя получить выигрыш в виде более быстрых операций

Вы можете также применить эту концепцию к более крупным хранилищам данных, точно так же, как и к реляционным наборам данных. Хитрость с индексами заключается в четком понимании того, как пользователи получают доступ к вашим данным. В случае если объемы наборов данных измеряются многими терабайтами, а полезной информации в них совсем немного (например, 1 Кбайт), использование индексов является необходимостью для оптимизации доступа к данным. Нахождение малой по размеру полезной информации в таком большом наборе данных может быть реальной проблемой, так как вы точно не сможете последовательно перебрать такое большое количество данных за любое разумное время. Кроме того, весьма вероятно, что такой большой набор данных распределен между несколькими (или многими!) физическими устройствами, и это означает, что вам необходимо каким-то образом найти правильное физическое местоположение нужных данных. Индексы - лучший способ сделать это.

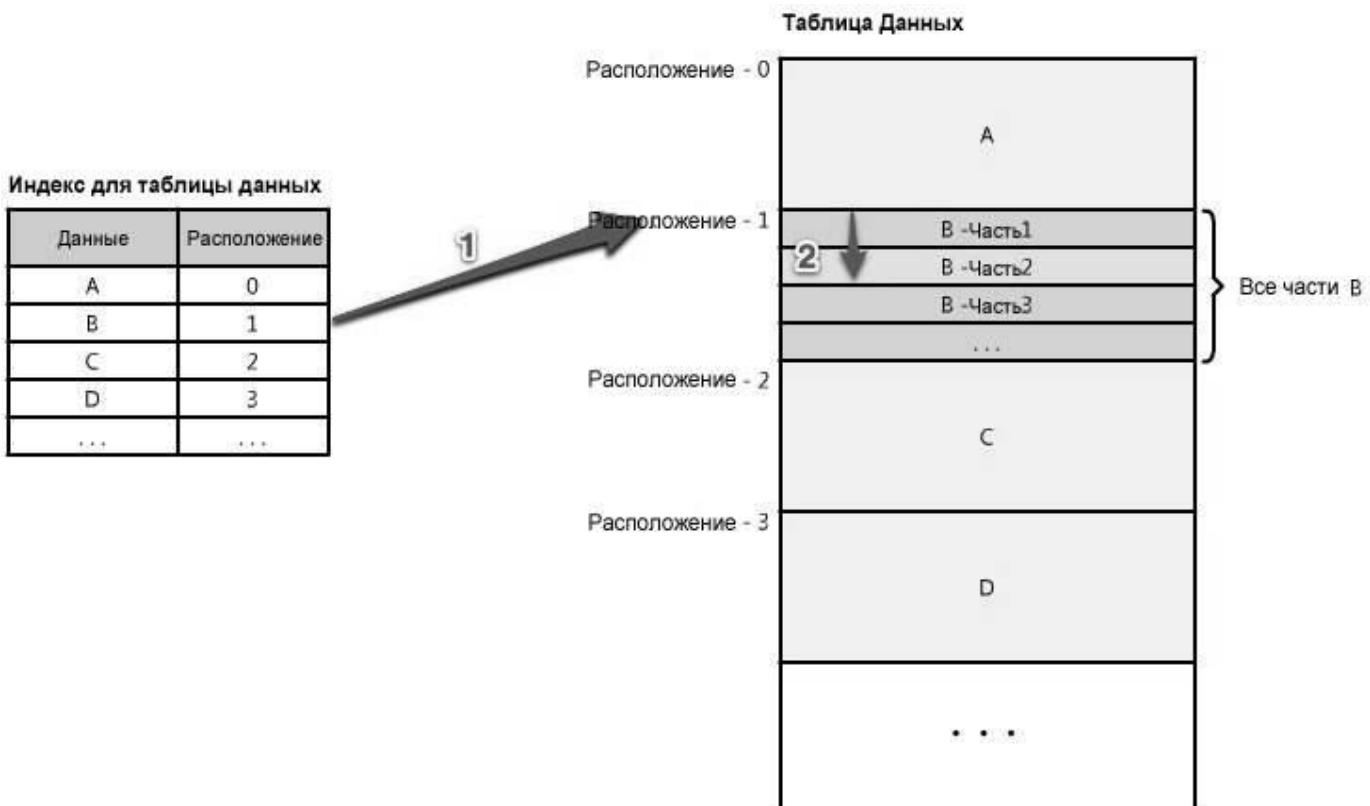


Рисунок 1.16: Индексы

Индекс может использоваться как оглавление, которое направляет вас к местоположению ваших данных. К примеру, скажем, вы ищете порцию данных, часть ©2 секции "B" - как вы узнаете, где ее найти? Если у вас есть индекс, отсортированный по типу данных - назовем данные "A", "B", "C" - он укажет вам расположение данных "B" в источнике. Тогда вы просто должны найти это расположение и считать ту часть "B", которая вам нужна. (См. рисунок 1.16)

Данные индексы часто хранятся в памяти или где-нибудь очень локально по отношению к входящему запросу клиента. Berkeley DB (BDB) и древовидные структуры данных, которые обычно используются, чтобы хранить данные в упорядоченных списках, идеально подходят для доступа с индексом.

Часто имеется много уровней индексов, которые служат картой, перемещая вас от одного местоположения к другому, и т.д., до тех пор пока вы не получите ту часть данных, которая вам необходима. (См. рисунок 1.17)

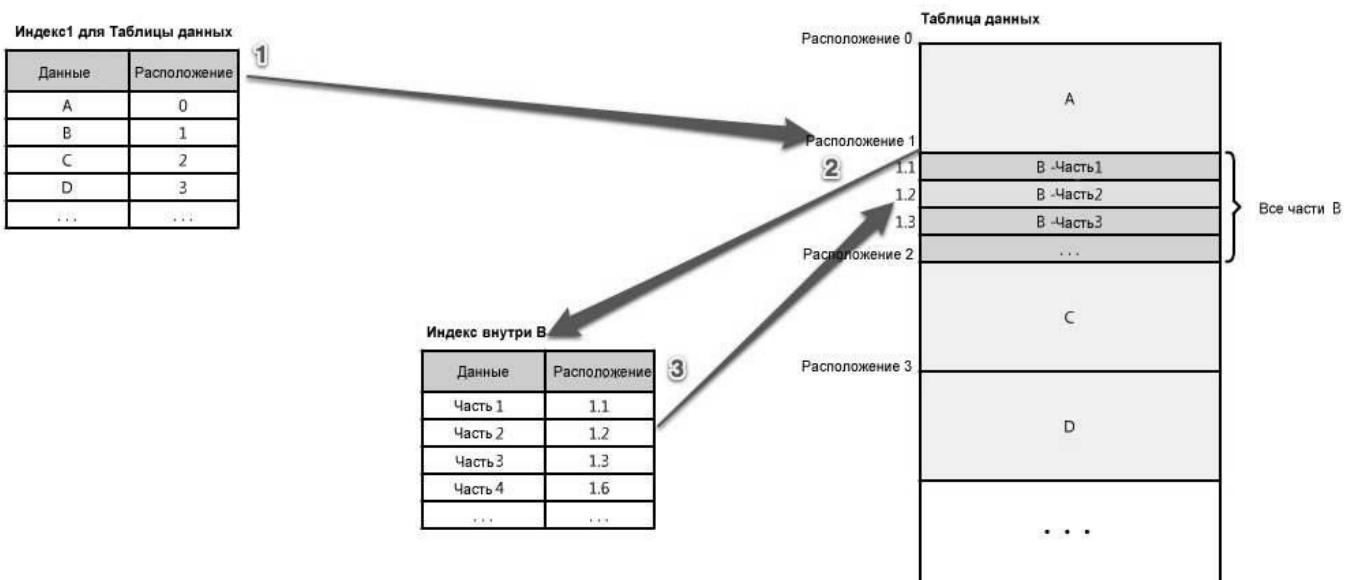


Рисунок 1.17: Многоуровневые индексы

Индексы могут также использоваться для создания нескольких других представлений тех же данных. Для больших наборов данных это - отличный способ определить различные фильтры и виды, не прибегая к созданию многих дополнительных копий данных.

Например, предположим, что система хостинга изображений, упомянутая выше, на самом деле размещает изображения книжных страниц, и сервис обеспечивает возможность клиентских запросов по тексту в этих изображениях, ища все текстовое содержимое по заданной теме также, как поисковые системы позволяют вам искать по HTML-содержимому. В этом случае все эти книжные изображения используют очень много серверов для хранения файлов, и нахождение одной страницы для представления пользователю может быть достаточно сложным. Изначально обратные индексы для запроса произвольных слов и наборов слов должны быть легкодоступными; тогда существует задача перемещения к точной странице и месту в этой книге и извлечения правильного изображения для результатов поиска. Таким образом, в этом случае инвертированный индекс отобразился бы на местоположении (таком как книга B), и затем B может содержать индекс со всеми словами, местоположениями и числом возникновений в каждой части.

Инвертированный индекс, который может отобразить Index1 в схеме выше, будет выглядеть при-

мерно так: каждое слово или набор слов служат индексом для тех книг, которые их содержат.

будучи удивительной	книга В, Книга С, Книга D
всегда	Книга С, Книга F
верьте	Книга В

Промежуточный индекс будет выглядеть похоже, но будет содержать только слова, местоположение и информацию для книги В. Такая содержащая несколько уровней архитектура позволяет каждому из индексов занимать меньше места, чем, если бы вся эта информация была сохранена в один большой инвертированный индекс.

И это ключевой момент в крупномасштабных системах, потому что даже будучи сжатыми, эти индексы могут быть довольно большими и затратными для хранения. Предположим, что у нас есть много книг со всего мира в этой системе, - 100,000,000 (см. запись блога "["Внутри Google Books"](#)")- и что каждая книга состоит только из 10 страниц (в целях упрощения расчетов) с 250 словами на одной странице: это суммарно дает нам 250 миллиардов слов. Если мы принимаем среднее число символов в слове за 5, и каждый символ закодируем 8 битами (или 1 байтом, даже при том, что некоторые символы на самом деле занимают 2 байта), потратив, таким образом, по 5 байтов на слово, то индекс, содержащий каждое слово только один раз, потребует хранилище емкостью более 1 терабайта. Таким образом, вы видите, что индексы, в которых есть еще и другая информация, такая, как наборы слов, местоположение данных и количества употреблений, могут расти в объемах очень быстро.

Создание таких промежуточных индексов и представление данных меньшими порциями делают проблему более простой в решении. Данные могут быть распределены на множестве серверов и в то же время быть быстродоступны. Индексы - краеугольный камень информационного поиска и база для сегодняшних современных поисковых систем. Конечно, этот раздел лишь в общем касается темы индексирования, и проведено множество исследований о том, как сделать индексы меньше, быстрее, содержащими больше информации (например, релевантность), и беспрепятственно обновляемыми. (Существуют некоторые проблемы с управляемостью конкурирующими условиями, а также с числом обновлений, требуемых для добавления новых данных или изменения существующих данных, особенно в случае, когда вовлечены релевантность или оценка).

Очень важна возможность быстро и легко найти ваши данные, и индексы - самый простой и эффективный инструмент для достижения этой цели.

1.4. Заключение

Разработка эффективных систем с быстрым доступом к большому количеству данных является очень интересной темой, и существует еще значительное число хороших инструментов, которые позволяют адаптировать все виды новых приложений. Эта глава коснулась всего лишь нескольких примеров, но в реальности их гораздо больше - и создание новых инноваций в этой области будет только продолжаться.

2. Технология подготовки релизов веб-браузера Firefox

Глава 2 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Недавно группа подготовки релизов организации Mozilla внесла множество улучшений в технологию автоматизации выпуска релизов веб-браузера Firefox. Мы сократили требования, предусматривающие человеческое вмешательство на этапах добавления подписей и отправки уведомлений о релизе заинтересованным сторонам, а также автоматизировали многие другие небольшие выполнявшиеся вручную этапы, так как каждый выполняющийся вручную этап процесса подготовки релиза может быть источником человеческой ошибки. Принимая во внимание то, что тот процесс, который мы используем на сегодняшний день, не является идеальным, мы при любой возможности стремимся дополнительно гладить и автоматизировать его. Нашей конечной целью является процесс, позволяющий нажать кнопку и продолжить заниматься своими делами; минимальное человеческое вмешательство позволит исключить многие проблемы и излишние действия, которые были свойственны для нашего старого частично ручного и частично автоматизированного процесса подготовки релизов. В данной главе мы изучим и опишем сценарии и принятые в отношении инфраструктуры решения, на основе которых была создана система ускоренного выпуска релизов веб-браузера Firefox, применяемая с момента выпуска Firefox 10.

Вы проследите процесс развития системы с момента появления пригодного для релиза набора изменений в системе контроля версий Mercurial до превращения этого набора в релиз-кандидат, а потом и в публичный релиз, доступный ежедневно для более чем 450 миллионов пользователей со всего мира. Мы начнем рассмотрение со сборки и добавления подписей к сборкам, после чего рассмотрим измененные партнерские и специально локализованные сборки, процесс контроля качества, а также способ генерации обновлений для каждой поддерживаемой версии, платформы и локализации. Каждый из этих этапов должен быть завершен до того момента, как релиз сможет быть размещен в сети зеркал организации Mozilla, с которых его смогут скачать наши пользователи.

Мы рассмотрим некоторые решения, которые были приняты для улучшения описанного процесса; например, наш сценарий для проверки работы функций в стандартных условиях, который помогает нам исправлять множество недоработок, заключающихся в неустойчивости программного продукта к пользовательским ошибкам; наш сценарий автоматического добавления подписи; нашу технологию интеграции процесса выпуска релизов для мобильных устройств в процесс выпуска релизов для настольных компьютеров; систему генерации патчей, в рамках которой создаются обновления, а также AUS (сервис обновления приложений - Application Update Service), благодаря которому обновления отправляются нашим пользователям, установившим различные версии программного обеспечения.

В этой главе подробно описывается процесс генерации сборок веб-браузера Firefox, которые становятся релизами. Большая часть этой главы посвящена подробному описанию важных этапов, которые встречаются в процессе подготовки релиза после запуска сборки, но также здесь описываются сложные взаимодействия, осуществляемые между рабочими группами до того, как группа подготовки релизов приступает к генерации сборок для релиза, поэтому давайте начнем с этого момента.

2.1. Рассмотрите N способов до момента подготовки релиза



Рисунок 2.1: Процесс подготовки релиза от разработки кода до команды "отправка на сборку"

В момент начала реализации проекта по усовершенствованию процесса выпуска релизов программных продуктов организации Mozilla, мы использовали в качестве исходного условия тот факт, что чем более популярным станет веб-браузер Firefox, тем у нас будет больше пользователей, а веб-браузер станет более привлекательной целью для злоумышленников, ищущих уязвимости системы безопасности для последующей эксплуатации. Также, чем более популярным станет веб-браузер Firefox, тем тем большее количество пользователей нам придется защищать от вновь и вновь появляющихся уязвимостей системы безопасности, поэтому наиболее важным аспектом становится возможность доставки исправлений для системы безопасности так быстро, как это возможно. Для такой ситуации у нас даже есть термин: "chemspill"-релиз (сокращение от "chemical spill" - "разлив химикатов"). Вместо редких выпусков chemspill-релизов в промежутках времени между регулярно планируемыми выпусками релизов, мы решили проводить планирование с учетом того, что каждый релиз может быть chemspill-релизом и спроектировали систему автоматизации в соответствии этой моделью планирования.

Данный подход имеет три важных последствия:

- Мы подводим итоги работы после выпуска каждого релиза и ищем области, в которых работа могла быть выполнена в следующий раз более плавно, просто и быстро. Если все это возможно, мы ищем и незамедлительно до выпуска следующего релиза исправляем как минимум одну недоработку, причем не важно насколько эта недоработка значительна. Постоянное усовершенствование нашей системы автоматизации выпуска релизов подразумевает то, что мы всегда ищем новые пути для того, чтобы снизить степень вовлеченности людей в процесс и в то же время улучшить устойчивость системы и сократить время выполнения работы. Большие усилия были затрачены на то, чтобы сделать наши инструменты безопасными, таким образом "редкие" события, например, нарушения работы сети, недостаток дискового пространства или опечатки, допущенные живыми людьми обнаруживаются и обрабатываются на таких ранних этапах, как это возможно. Несмотря на то, что наша система на данный момент достаточно быстра для выпуска обычных релизов, не являющихся chemspill-релизами, мы хотим сократить риск появления в будущем релизе любой ошибки, допущенной человеком. Это особенно актуально для chemspill-релизов.
- При подготовке chemspill-релиза мы исходим из того, что чем устойчивее система автоматизации, тем меньше подвержены стрессу люди из группы подготовки релизов. Мы придерживаемся идеи, заключающейся в выполнении работы так быстро, как это возможно в спокойной обстановке и мы создали инструменты для выполнения работы в настолько безопасном и надежном режиме, насколько позволяют наши знания. Меньшая подверженность стрессу подразумевает более спокойную и тщательную работу в ходе выполнения хорошо отрепетированного процесса, что в свою очередь помогает избежать сложностей при выпусках chemspill-релизов.
- Мы создали процесс "отправки на сборку" в масштабах организации Mozilla. При подготовке обычного (не chemspill) релиза каждому участнику процесса разработки может быть предоставлена возможность обзора одних и тех же отсортированных цепочек описаний ошибок, точного установления момента применения последнего исправления и удачного тестирования, а также достижения договоренности о том, когда должны быть начаты сборки. Однако, в случае chemspill-релиза, когда минуты имеют значение, отслеживание всех особенностей ошибки вместе с чтением всех последующих подтверждений наличия ошибки и предложенных исправлений через некоторое время становится очень сложным занятием. Для снижения сложности и риска совершения ошибок организация Mozilla наняла человека, который в течение полночь рабочего дня отслеживает готовность кода для "отправки на сборку". Изменение процессов в ходе выпуска chemspill-релиза рискованно, поэтому для уверенности в том, что каждый знаком с процессом выпуска релиза в ситуации, когда минуты имеют значение, мы используем один и тот же процесс подготовки к выпуску chemspill- и обычных релизов.

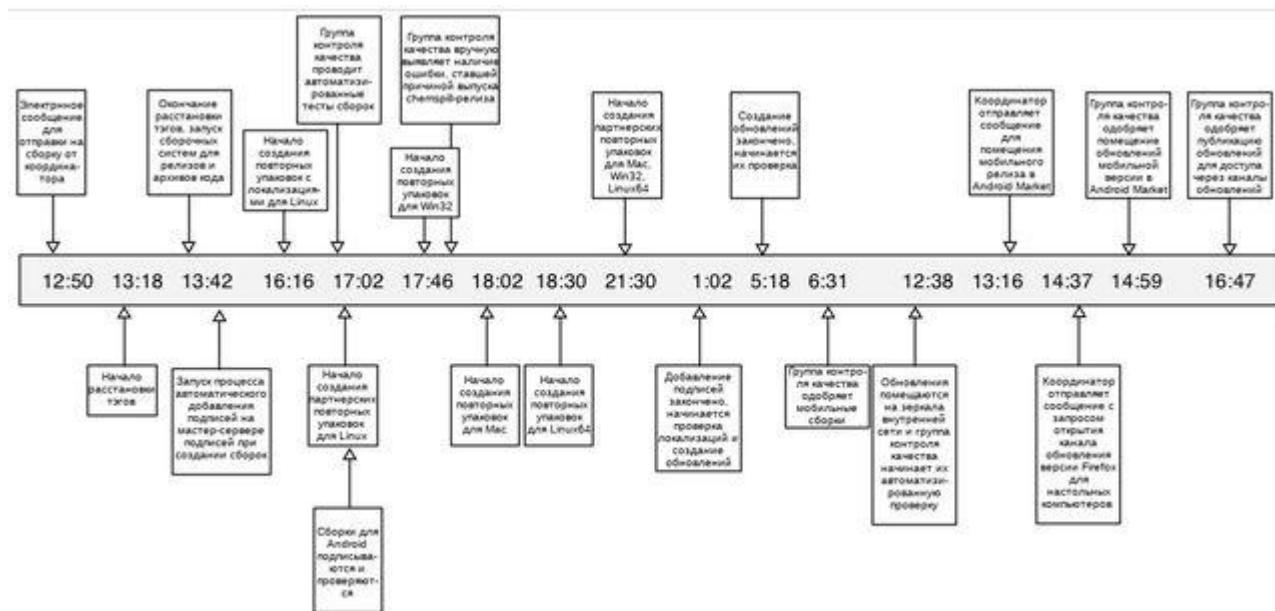


Рисунок 2.2: Полное распределение по времени этапов процесса подготовки chemspill-релиза, используемого в качестве примера

2.2. "Отправка на сборку"

Кто может быть источником команды "отправка на сборку"?

Перед началом подготовки релиза назначается один человек, который будет нести ответственность за координацию всего процесса выпуска релиза. Этот человек должен посещать собрания, на которых координируется работа, понимать контекст всей выполненной работы, справедливо рассуждать о приоритетах ошибок, подтверждать возможность внесения запоздавших изменений, а также принимать строгие решения, заключающиеся в отклонении предложений. К тому же, непосредственно в день релиза этот человек является центром взаимодействия различных групп (разработчиков, контроля качества, подготовки релизов, разработчиков сайта, по связям со СМИ, по маркетингу, и.т.д.).

Различные компании используют различные термины для обозначения этой должности. Среди некоторых услышанных нами терминов есть такие, как менеджер релизов (Release Manager), инженер релизов (Release Engineer), менеджер программ (Program Manager), менеджер проекта (Project Manager), менеджер продукта (Product Manager), руководитель выпуска продукта (Product Czar), руководитель релиза (Release Driver). В данной главе мы будем использовать термин "координатор релиза" ("Release Coordinator"), так как мы считаем, что он наиболее точно отражает соответствующую роль в описанном выше процессе. Важным моментом является то, что роль и конечные полномочия выступающего в данной роли человека должны быть четко поняты всеми участниками процесса перед началом подготовки релиза вне зависимости от любых предшествующих периодов совместной работы над каким-либо проектом. В напряженной обстановке, создающейся в день релиза, важно, чтобы каждый знал, что от этого человека следует ожидать принятия решений, связанных с координацией процесса, придерживаться их и доверять им.

Координатор релиза является единственным человеком вне группы подготовки релизов, который может отправлять электронные письма для выполнения "остановки сборок" в том случае, если обнаруживается проблема, не позволяющая распространять собранный релиз. Любые отчеты о подозрительных проблемах, которые также могут препятствовать распространению релиза, перенаправляются координатору релиза, который впоследствии исследует их, примет окончательное решение о их принятии или отклонении и своевременно сообщит об этом решении всем заинтересованным сторонам. В напряженной обстановке момента сборки релиза нам всем следует ожидать принятия этим человеком связанных с координацией процесса сборки решений, придерживаться их и доверять им.

Как передать команду "отправка на сборку"?

Ранние эксперименты с передачей команды "отправка на сборку" посредством IRC-каналов или устной передачей этой же команды по телефону приводили к непониманию со стороны участников команды, которое порой порождало проблемы в процессе выпуска релиза. Исходя из этого, на данный момент мы требуем передачи сигнала "отправка на сборку" для каждого релиза с использованием электронной почты путем отправки сообщения в список рассылки, на который подписаны участники всех групп, занимающиеся выпуском релизов. Тема электронного сообщения должна включать фразу "go to build" и четко указанное название продукта с номером версии, например:

```
go to build Firefox 6.0.1
```

Аналогично, в том случае, если в релизе обнаруживается проблема, координатор релиза передаст команду "остановить все сборки" ("all stop") в форме электронного сообщения с новой темой, отправленного в тот же список рассылки. Мы посчитали неэффективной отправку всего лишь ответа на последнее относящееся к релизу электронное сообщение; в некоторых клиентах электронной почты темы обсуждений оформляются таким образом, что пользователи не заметят сообщение с командой "остановить все сборки", так как оно будет находиться гораздо ниже в несвязанной теме.

Какая информация содержится в электронном сообщении с командой "отправка на сборку"?

1. Указание на именно тот исходный код, который должен использоваться для релиза; в идеальном случае это указание должно быть строкой URL, указывающей на определенное изменение в репозитории исходного кода, на основе которого будут формироваться сборки для релиза.
 1. Инструкции, аналогичные "использованию новейшего исходного кода" неприемлемы ни в каком случае; при подготовке одного из релизов в момент после передачи команды "отправка на сборку" в сообщении электронной почты и до непосредственного начала процесса сборки разработчик из лучших побуждений разместил неподтвержденное изменение исходного кода в не предназначенной для изменения ветви репозитория. Это незапланированное изменение вошло в релиз. Благодаря нашей внимательности ошибка была выявлена до момента предоставления публичного доступа к релизу, тем не менее, нам пришлось задержать релиз из-за затрат времени на полную остановку процесса сборки и на выполнение повторной сборки.

2. При использовании подобной CVS системы контроля версий, работающей на основе меток времени, следует явно указывать точное время; указывайте время с точностью до секунд и добавляйте к нему указание часового пояса. Во время выпуска одного из релизов в то время, когда для хранения кода Firefox все еще использовалась система CVS, координатор релиза указал граничное время изменений кода для сборки, но не указал часовой пояс. Со временем группа подготовки релизов обнаружила отсутствие информации о часовом поясе, но координатор релиза уже спал. Участники группы подготовки релизов правильно догадались, что должно было использоваться местное время (для штата Калифорния), но из-за ночной путаницы и использования часового пояса PDT вместо PST мы потеряли последнее исправление критической ошибки. Эта недоработка была выявлена группой контроля качества продукта перед предоставлением доступа к сборке широкому кругу пользователей и нам также пришлось останавливать сборку и начинать ее сначала с использованием корректного граничного времени изменений кода.
2. Четкое обоснование срочности данного релиза. Хотя эта информация и кажется очевидной, она важна при работе в некоторых нестандартных ситуациях, поэтому ниже приведено краткое описание типов релизов:
 1. Некоторые релизы являются "обычными" релизами и могут подготавливаться в обычное рабочее время. Это ранее запланированные релизы, которые выпускаются в оговоренное время и не требуют выполнения срочной работы. Конечно же, все сборки программных продуктов для релизов должны формироваться своевременно, но от людей, занимающихся подготовкой релизов, не требуется работы в течение нескольких ночей и приложения всех сил для выпуска обычного релиза. Вместо этого мы заранее правильно планируем выпуск таких релизов, поэтому весь процесс подготовки релиза проходит в соответствии с ранее составленным графиком и люди работают над релизом в обычное рабочее время. Такой подход позволяет не загружать людей излишней работой, причем они смогут работать и в случае появления необходимости выполнения срочной незапланированной работы.
 2. Некоторые релизы являются chemspill-релизами, которые должны срочно подготавливаться и выпускаться в условиях, когда каждая минута на счету. Эти релизы обычно выпускаются в случае необходимости исправления ошибки, используемой опубликованным экспloitом или в случае необходимости исправления недавно выявленной ошибки, заключающейся в аварийном завершении процесса и затрагивающей большую часть нашей пользовательской базы. Chemspill-релизы должны создаваться так быстро, как это возможно и обычно при их выпуске не используется предварительное планирование.
 3. Некоторые релизы могут превращаться из обычных релизов в chemspill-релизы и наоборот из chemspill-релизов в обычные релизы. Например, в том случае, если исправление безопасности из обычного релиза было непредумышленно опубликовано, обычный релиз превращается в chemspill-релиз. В том случае, если маркетинговое требование, заключающееся в выпуске "предварительного специального релиза" для демонстрации на проходящей конференции приводит к задержке выпуска релиза по маркетинговым соображениям, релиз превращается из chemspill-релиза в обычный релиз.
 4. По отношению к срочности некоторых релизов разные люди могут иметь разные мнения, зависящие от их взгляда на исправления, поставляемые в рамках релиза.

В обязанности координатора релиза входит взвешивание всех фактов и мнений и вынесение решения о срочности релиза, а также взаимодействие со всеми группами и сообщение о принятом решении. В случае получения новой информации координатор релиза пересматривает решение и затем снова сообщает о новой установленной срочности релиза участникам тех же групп. В том случае, если участники некоторых групп будут считать релиз chemspill-релизом и в то же время участники других групп будут считать тот же релиз обычным релизом, взаимодействие групп может быть нарушено.

Наконец, эти сообщения электронной почты также становятся очень полезными в случае необходимости измерения времени, прошедшего с момента выпуска релиза. Хотя точность измерения времени соответствует точности обычных часов, эта информация очень полезна при установлении той области, в которой в следующий раз нужно приложить наши силы для ускорения процесса. Как говорит древняя поговорка, перед тем, как вы сможете улучшить что-либо, у вас должна быть возможность измерить это.

В течение цикла бета-тестирования браузера Firefox мы также еженедельно выпускаем релизы на основе кода из нашего [репозитория mozilla-beta](#). Каждый из этих бета-релизов проходит обычные стадии подготовки в нашей полностью автоматизированной системе и рассматривается практически идентично нашим обычным финальным релизам. Для уменьшения количества неприятных сюрпризов в процессе подготовки релиза мы пытаемся не вносить новых не протестированных изменений в систему автоматизации выпуска релизов или инфраструктуру до момента начала сборки финального релиза.

2.3. Использование тэгов, сборка и архивы с исходным кодом

Репозитории инструментов для внутреннего использования и конфигурационных файлов с версиями для того, чтобы мы могли выяснить, какая система автоматизации используется

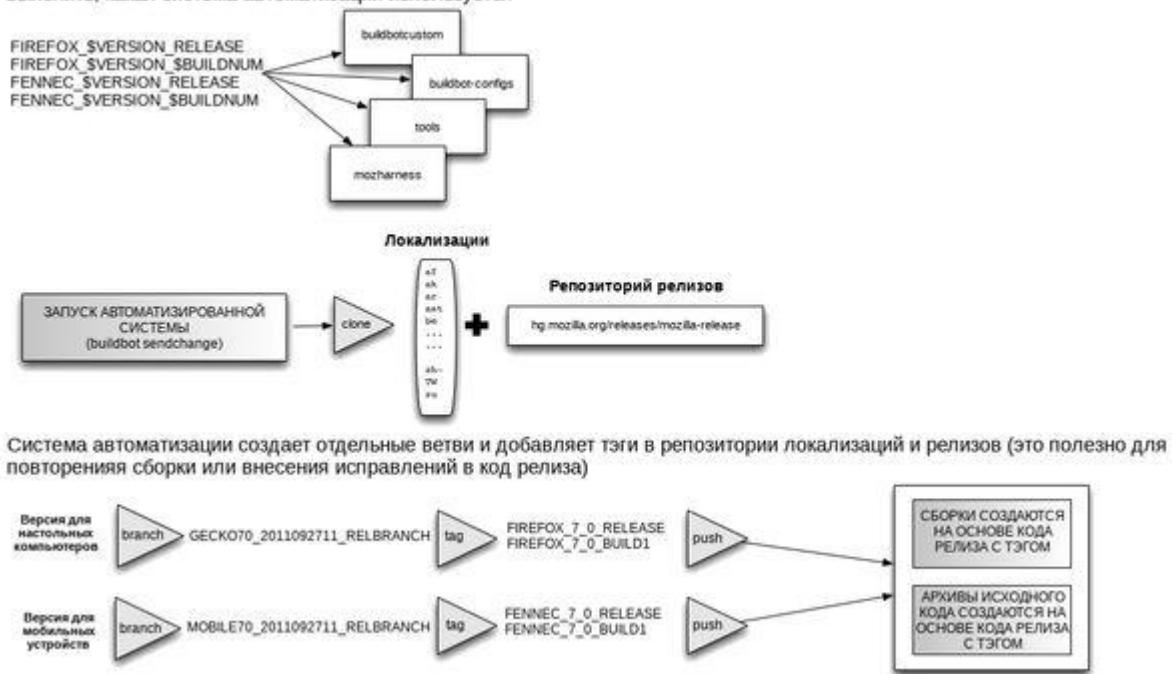


Рисунок 2.3: Автоматизированная расстановка тэгов

При подготовке к началу процесса автоматизации системы мы приступили к использованию [сценария release_sanity.py](#), который был разработан интерном, проходящим летнюю стажировку в группе подготовки релизов. Этот сценарий на языке Python помогает подготавливающему релиз участнику группы, производя двойную проверку соответствия конфигурации релиза конфигурациям инструментов и репозиториев. Он также устанавливает выбранные ревизии исходного кода для релиза из ветви `mozilla-release` и все поставляемые в рамках этого релиза (человеческие) языки, на основе которых будут сгенерированы сборки для выпуска релиза и пересборки с поддержкой определенных языков.

Сценарий принимает файлы конфигурации системы непрерывной интеграции `buildbot` для любых конфигураций релиза, которые будут использоваться (таких, как конфигурации для сборки версий для настольных компьютеров и мобильных устройств), указание на ветвь в репозитории исходного кода для поиска (т.е., `mozilla-release`), номер сборки и версию, а также названия продуктов, которые будут собраны (такие, как "fennec" или "firefox"). Его выполнение завершится неудачей в том случае, если конфигурации репозиториев кода для формирования релизов не совпадают с заданной конфигурацией, если наборы изменений в репозитории файлов локализации не совпадают с поставляемыми нами наборами изменений файлов локализации или в том случае, когда версия и номер сборки не совпадают с переданными нашим инструментам сборки параметрами в форме тэга на основе номеров продукта, версии и сборки. В том случае, если все проверки из сценария завершатся успешно, он произведет повторную конфигурацию мастер-сервера системы `buildbot`, на котором выполняется сценарий и который будет управлять системами сборки релизов, после чего генерирует команду "отправка изменений" ("send change"), которая запускает автоматизированный процесс подготовки сборки.

После того, как участник группы подготовки релиза запускает системы для сборки, первым автоматически выполняемым шагом в процессе подготовки релиза Firefox является установка тэга во всех используемых репозиториях исходного кода для записи номера ревизии данных из репозиториев исходного кода, языков и связанных инструментов, которые были использованы для подготовки данной версии и сборки релиз-кандидата. Эти тэги позволяют нам отслеживать историю версий и номеров сборок релизов Firefox и Fennec (мобильной версии Firefox) в рамках наших репозиториев для релизов. Для релизов Firefox примером установленного тэга может служить `FIREFOX_10_0_RELEASE FIREFOX_10_0_BUILD1 FENNEC_10_0_RELEASE FENNEC_10_0_BUILD1`.

При подготовке каждого отдельного релиза Firefox используется код в среднем из 85 репозиториев системы контроля версий, которые хранят такие данные, как код программного продукта, строки локализации, код системы автоматизации для выпуска релизов, а также код вспомогательных утилит. Присвоение тэгов коду из всех этих репозиториев критически важно, так как необходима гарантия того, что на следующих этапах работы системы автоматизации выпуска релизов будут выполняться действия с тем же набором ревизий данных.

Такой подход также имеет ряд других преимуществ: команды разработчиков дистрибутивов Linux и другие участники процесса разработки могут самостоятельно повторять процесс сборки, используя тот же самый код и инструменты, что и в случае подготовки официальных сборок, а также производится запись ревизий исходного кода продукта и инструментов, используемых в ходе подготовки каждого релиза, на основе данных которой в будущем возможно сравнение изменений, осуществленных между релизами.

Как только во всех репозиториях выделены ветви и созданы тэги, группа зависимых сборочных систем автоматически начинает работу: выделяется по одной сборочной системе для каждой платформы, а также для подготовки архива исходного кода, включающего весь исходный код, используемый в релизе. Архив исходного кода и собранные установщики загружаются в директорию для релизов по мере доступности. Это позволяет любому человеку ознакомиться с тем, какой именно код был использован для формирования релиза, а также появляется возможность использовать архив исходного кода для повторной сборки в том случае, когда появится такая необходимость (например, в случае какого-либо сбоя нашей системы контроля версий). Для создания архива исходного кода Firefox нам иногда приходится импортировать код из репозитория, предназначенному для подготовки более ранних версий. Например, в случае выпуска бета-релиза происходит извлечение подписанной ревизии из репозитория Mozilla-Autoga (нашего репозитория исходного кода, отличающегося большей стабильностью, чем репозиторий для подготовки ночных сборок) для версии Firefox 10.0b1.

В случае выпуска релиза происходит перемещение подтвержденных изменений из репозитория Mozilla-Beta (практически с тем же кодом, который использовался для подготовки версии 10.0b6) в репозиторий Mozilla-Release. Позднее ветвь для выпуска релиза создается в форме именованной ветви, родительским набором изменений которой является подписанная ревизия, предназначенная для "отправки на сборку" и предоставленная координатором релиза. Ветвь для выпуска релиза может быть использована для выполнения специфичных для релиза модификаций исходного кода, таких, как увеличение номеров версий или завершение формирования списка локализаций, которые будут собраны. В будущем при обнаружении критической проблемы безопасности и необходимости ее немедленного исправления, на основе этой ветви для выпуска релиза может быть сформирован содержащий минимальное количество изменений для исправления уязвимости chemspill-релиз, после чего будет сгенерирована и выпущена новая версия Firefox.

Когда нам понадобится произвести второй цикл сборок определенного релиза под названием buildN, мы используем эти ветви для выпуска релиза с целью получения того же кода, который был подписан для "отправки на сборку", при этом любые изменения в коде релиза будут размещаться именно в этой ветви. Автоматизированный процесс сборки начинается снова с создания тэгов для нового набора изменений в ветви для подготовки релиза. В ходе выполнения нашего процесса присвоения тэгов производится большое количество операций с локальными и удаленными репозиториями исходного кода системы контроля версий Mercurial. Для упрощения некоторых из наиболее часто выполняемых операций мы разработали несколько вспомогательных инструментов: `retry.py` и `hgtool.py`. Сценарий `retry.py` является простой оболочкой, которая получает переданную команду и выполняет ее, пытаясь в течение нескольких раз повторить выполнение в случае неудачи. Он также следит за возникновением исключительных условий при выводе результатов выполнения команд и повторно выполняет команды, либо выводит сообщения в подобных случаях. Мы посчитали удобным решением выполнение при посредничестве сценария

`retry.py` большей части команд, которые могут завершиться неудачей из-за внешних зависимостей.

При присваивании тэгов операции системы контроля версий Mercurial могут завершиться неудачей в случае временной неработоспособности сети, неработоспособности веб-серверов или временной чрезмерной нагрузки на используемый сервер системы Mercurial. Возможность автоматического повтора этих операций и продолжения выполнения работы сохраняет очень много нашего времени, так как нам не приходится вручную восстанавливать и удалять результаты неудавшейся попытки сборки, после чего снова запускать автоматизированную систему подготовки релизов.

Сценарий `hgtool.py` является утилитой, инкапсулирующей некоторые стандартные операторы системы Mercurial, такие, как операторы клонирования, операторы извлечения и обновления данных репозитория, при этом позволяющей выполнять их в ходе однократного запуска. Он также добавляет поддержку расширения для разделения данных системы Mercurial, которое широко используется нами для предотвращения появления нескольких клонированных копий репозиториев в различных директориях одной и той же машины. Добавление поддержки разделяемых локальных репозиториев значительно ускорило наш процесс присвоения тэгов, так как большинство полностью клонированных репозиториев с кодом продукта и данными локализации могли быть исключены из процесса. Важной мотивацией для разработки подобных инструментов является предоставление возможности тестирования системы автоматизации настолько, насколько это возможно. Так как такие инструменты, как `hgtool.py` являются небольшими утилитами для выполнения единственной задачи, созданными на основе многократно используемых библиотек, их гораздо проще тестировать в изолированном окружении.

На сегодняшний день наш процесс присвоения тэгов состоит из двух параллельно выполняемых процессов: первым является процесс присвоения тэгов версии Firefox для настольных компьютеров, который длится около 20 минут, так как предусматривает присвоение тэгов данным в более чем 80 репозиториях локализаций, а вторым является процесс присвоения тэгов данным мобильной версии Firefox, который занимает около 10 минут, так как для мобильных релизов на данный момент доступно меньшее количество локализаций. В будущем мы хотели бы ускорить наш процесс автоматической подготовки релизов, поэтому мы хотим присваивать тэги данным во всех различных репозиториях в параллельном режиме. Начальная сборка программных продуктов может запускаться тогда, когда данным в репозиториях с кодом продукта и кодом требуемых для сборки инструментов будут присвоены тэги без необходимости ожидания присваивания тэгов данным во всех репозиториях локализаций. В момент завершения этих сборок данным в остальных репозиториях будут также присвоены тэги, поэтому пакеты локализаций смогут быть подготовлены и следующие шаги процесса сборки смогут завершиться. Мы ожидаем, что эти мероприятия помогут уменьшить общее время подготовки сборок примерно на 15 минут.

2.4. Повторно упакованные сборки с локализацией и партнерские повторно упакованные сборки

Как только сборки для настольных компьютеров генерируются и загружаются на сервер `ftp.mozilla.org`, наша автоматизированная система подготовки релизов переходит к выполнению работ по созданию повторно упакованных сборок с локализациями. "Повторно упакованная сборка с локализацией" формируется на основе оригинальной сборки (которая содержит локализацию `en-US`) путем ее распаковки, замены строк локализации `en-US` на строки другой локализации, которую мы будем распространять в рамках данного релиза, с последующей повторной упаковкой всех извлеченных файлов (именно поэтому мы и называем эти сборки повторно упакованными). Мы повторяем эту процедуру для каждой локализации, поставляемой в рамках релиза. Изначально мы осуществляли все повторные упаковки файлов последовательно. Однако, по мере того, как мы добавляли дополнительные локализации, этот процесс все больше затягивался и нам

приходилось начинать его выполнение с начала в случае возникновения ошибки на промежуточном этапе.

Расположение информации о более чем 80 локализациях Firefox

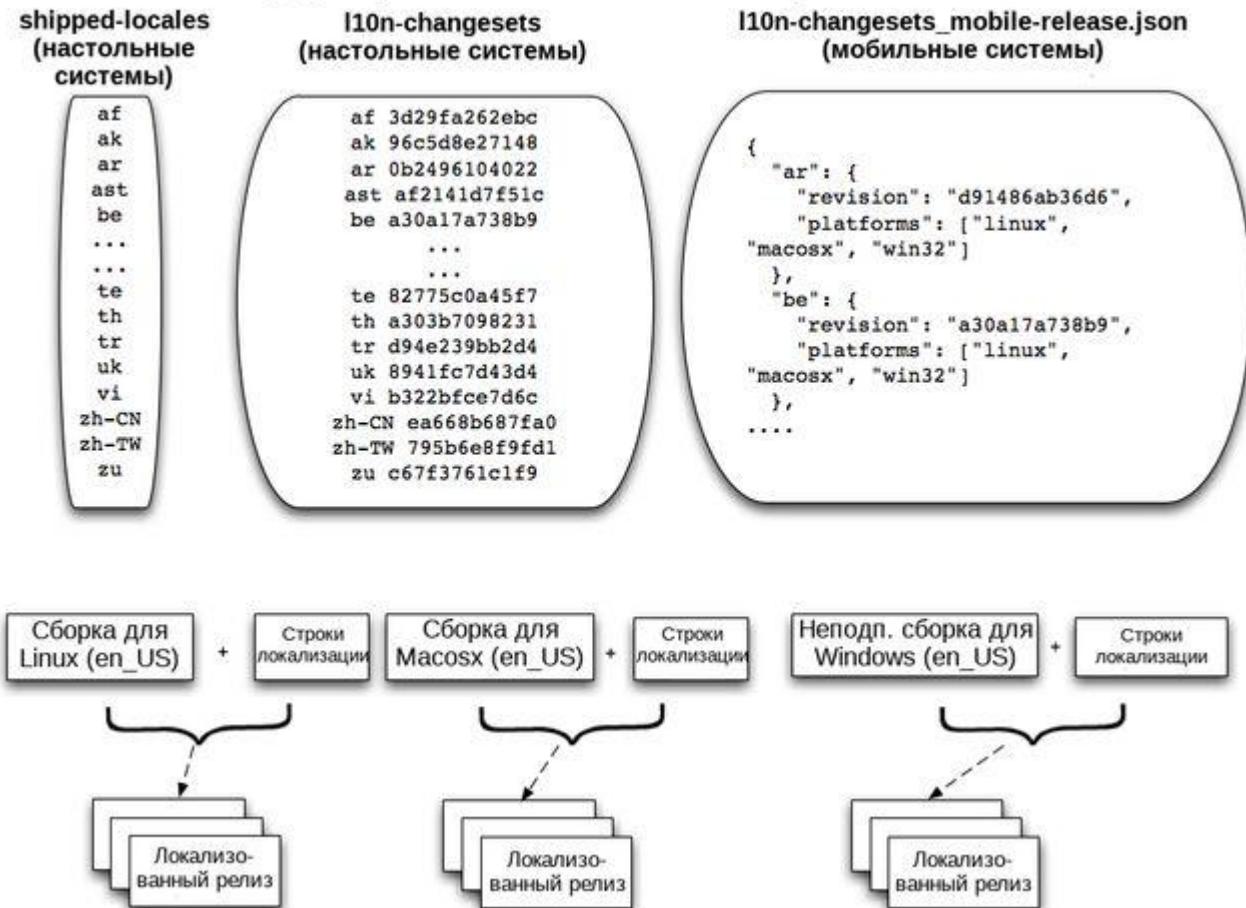


Рисунок 2.4: Повторная упаковка файлов Firefox для каждой из локализаций

В настоящее время мы разделили весь процесс повторной упаковки на шесть отдельных процессов, которые выполняются одновременно на шести отдельных машинах. Этот подход позволил сократить время работы практически в шесть раз. Также он позволил отменять повторную упаковку только ограниченного количества сборок в случае возникновения ошибки при упаковке без необходимости отмены всех упаковок. (Мы можем разделить задачи по выполнению повторных упаковок сборок даже на еще большее количество меньших одновременно обрабатываемых групп, но мы считаем, что такой подход подразумевает задействование слишком большого количества доступных машин, которые выполняют другие не связанные с повторной упаковкой сборок задачи, инициируемые разработчиками для обслуживания нашей системы непрерывной интеграции.)

Процесс для мобильной версии (предназначенной для работы под управлением Android) значительно отличается от описанного выше, так как мы выпускаем только два установщика: установщик англоязычной версии и установщик многоязычной версии со встроенной в установщик поддержкой множества языков вместо предоставления сборки для каждой локализации. Размер этой многоязычной версии имеет значение, особенно в случае использования медленных соединений и мобильных устройств с ограниченными ресурсами. Наш план на будущее заключается в предоставлении поддержки других языков посредством ресурса addons.mozilla.org по запросу.

На [Рисунке 2.4](#) вы можете увидеть, что на данный момент мы используем три различных источника информации о доступных локализациях: файлы `shipped_locales`, `l10n_changesets` и `l10n_changesets_mobile-release.json`. (Планируется перемещение данных из всех трех источников в

один файл в унифицированном формате JSON.) Эти файлы содержат информацию о различных локализациях, находящихся в нашем распоряжении, а также о исключениях для определенных платформ. В частности, при использовании заданной локализации нам необходимо обладать информацией о том, какая ревизия данных из репозитория должна быть использована для подготовки заданного релиза, а также нам необходимо обладать информацией о том, может ли локализация использоваться на всех поддерживаемых нами платформах (например, все данные японских локализаций для платформы Mac извлекаются из отдельного репозитория). Два из трех описанных выше файлов используются при подготовке релизов для настольных компьютеров и один файл - при подготовке релизов для мобильных устройств (в этом файле формата JSON содержится как список платформ, так и описание наборов изменений данных репозиториев).

Кто же принимает решение о том, какие языки мы включаем в комплект поставки релизов? Во-первых, создатели локализаций самостоятельно предлагают свои определенные наборы изменений локализаций для включения в состав заданного релиза. Предложенный набор изменений проверяется участниками команды локализации организации Mozilla и размещается на веб-ресурсе со списком наборов изменений, необходимых для поддержки каждого из языков. Координатор релиза проверяет размещенную на этом ресурсе информацию перед передачей сообщения электронной почты с командой "отправка на сборку". В день выпуска релиза мы извлекаем этот список наборов изменений и производим повторную упаковку сборок в соответствии с ним.

Наряду с повторными упаковками сборок с локализациями мы также генерируем партнерские повторно упакованные сборки. Это специально измененные сборки для различных партнеров, которые желают внести изменения в пользовательские качества программного продукта для удовлетворения запросов своих клиентов. Наиболее часто вносимыми типами изменений являются измененные списки закладок, нестандартные домашние страницы и нестандартные поисковые машины, но многие другие вещи также могут быть изменены. Эти измененные сборки генерируются для конечных релизов Firefox и не генерируются для бета-версий.

2.5. Добавление цифровых подписей

Для того, чтобы наши пользователи могли быть уверены в том, что они на самом деле скачали не модифицированную сборку от организации Mozilla, мы добавляем к сборкам несколько различных типов цифровых подписей.

Первый тип подписи используется для наших сборок, предназначенных для работы под управлением Windows. Мы используем ключ Microsoft Authenticode (технологии создания цифровой подписи на основе кода подписи) для добавления электронных подписей всем файлам с расширениями .exe и .dll. Windows может использовать эти подписи для проверки факта получения приложения из доверенного источника. Мы также подписываем исполняемый файл установщика Firefox с помощью ключа технологии Authenticode.

После этого мы используем GPG для генерации наборов контрольных сумм MD5 и SHA1 для всех сборок на всех платформах и генерируем отдельные подписи GPG для файлов контрольных сумм, а также для всех сборок и установщиков. Эти подписи используются зеркалами и участниками сообщества для проверки скачанных файлов.

В целях безопасности мы подписываем файлы на удаленной машине для генерации цифровых подписей, которая защищена от сторонних соединений с помощью межсетевого экрана и VPN. Наши электронные ключи, пароли и связки электронных ключей передаются между участвующими в подготовке релиза лицами исключительно посредством защищенных каналов, обычно персонально для возможной минимизации риска их раскрытия.

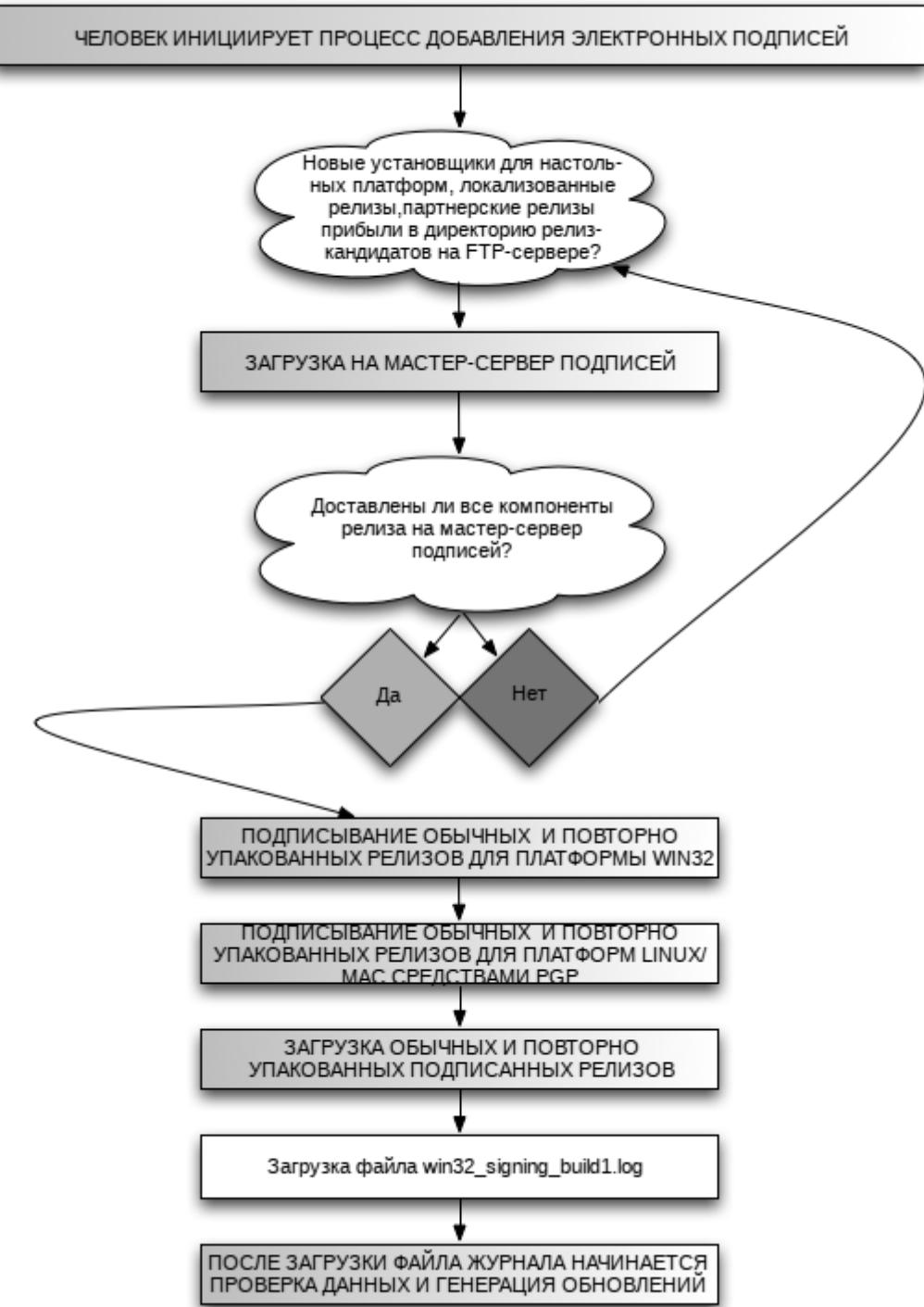


Рисунок 2.5: Добавление электронной подписи для установщиков Firefox

До недавнего времени процесс добавления электронных подписей требовал работы ответственного за выпуск релиза лица на удаленном сервере ("мастер-сервере электронных подписей"), которая занимала практически час и заключалась в ручной загрузке сборок, подписывании их и загрузке обратно на сервер <ftp.mozilla.org> перед тем, как автоматизированная система подготовки релиза сможет продолжить работу. После окончания процесса добавления электронных подписей на мастер-сервере и загрузки всех файлов, файл журнала осуществленных в ходе добавления подписей операций загружается в директорию для хранения релиз-кандидатов на сервере <ftp.mozilla.org>. Наличие этого файла журнала на сервере <ftp.mozilla.org> указывает на окончание ручной работы по добавлению подписей и с этого момента зависимые системы для сборки, которые следят за этим файлом, могут продолжить работу в автоматическом режиме. Не так давно мы добавили дополнительный уровень автоматизации для выполнения этапов добавления цифровых подписей. Теперь ответственное за выпуск релиза лицо может открыть командную оболочку

Cygwin на мастер-сервере электронных подписей и установить несколько переменных окружения, имеющих отношение к релизу, таких, как VERSION, BUILD, TAG и RELEASE_CONFIG, которые позволяют сценарию найти соответствующие директории на сервере <ftp.mozilla.org> и получить информацию о том, когда все сборки релиза будут загружены, после чего процесс добавления электронных подписей может начаться. После получения новейшей пригодной для эксплуатации версии наших инструментов для добавления электронных подписей подписывающий сборки человек может просто выполняет команду `make autosign`. После этого он вводит две ключевые фразы, одна из которых предназначается для gpg, а вторая - для кода подписи. Сразу после завершения автоматической проверки этих ключевых фраз средствами сценариев начинает выполняться автоматизированный цикл загрузки, в рамках которого производится наблюдение за загружаемыми сборками и повторно упакованными сборками, которые автоматически загружаются непосредственно после того, как становятся доступны. Сразу же после загрузки автоматизированная система начинает добавление цифровых подписей без необходимости человеческого вмешательства в процесс.

Отсутствие необходимости в человеческом вмешательстве при добавлении электронных подписей к сборкам важно по двум причинам. Во-первых, это обстоятельство позволяет сократить риск возникновения ошибки по вине человека. Во-вторых, это обстоятельство позволяет добавлять электронные подписи в нерабочее время, поэтому ответственному за выпуск релиза лицу не придется находиться за компьютером в неудобное для работы время.

Все сборки имеют соответствующие им файлы контрольных сумм MD5SUM и SHA1SUM, сгенерированные для них, причем значения контрольных сумм записываются в файлы с теми же именами, что и сборки. Эти файлы будут загружены назад в директорию для хранения релиз-кандидатов, а также будут скопированы в конечную директорию для хранения релизов на сервере <ftp.mozilla.org>, когда она будет создана, поэтому любой скачавший установщик Firefox с одного из наших зеркал человек может удостовериться в том, что он получил не модифицированный программный продукт. Когда все подписанные данные становятся доступны и проверяются, они загружаются назад на сервер <ftp.mozilla.org> вместе с файлом журнала операций добавления подписей, который ожидает система автоматизированной подготовки релиза.

Наш следующий запланированный этап усовершенствования процесса добавления электронных подписей заключается в создании инструмента, который позволит нам подписывать файлы одновременно с подготовкой сборки или повторной упаковкой сборки. Эта работа требует создания приложения для сервера электронных подписей, которое сможет принимать запросы на добавление подписей для файлов на машинах, осуществляющих сборку для выпуска релиза. Она также требует создания клиентского инструмента для работы с электронными подписями, который сможет соединяться с сервером электронных подписей, проходить аутентификацию, представляясь клиентом доверенной машины, которая в свою очередь может осуществлять запросы на добавление электронных подписей, ожидать добавления подписи, загружать подписанные данные и после этого включать их в состав сборки путем упаковки. Как только эти усовершенствования будут доступны для использования, мы сможем уйти от нашего последовательного процесса добавления всех электронных подписей, а также от нашего последовательного процесса генерации обновлений (о котором будет написано ниже). Мы ожидаем, что после выполнения этой работы общее время, затрачиваемое на подготовку релиза, сократится на несколько часов.

2.6. Обновления

Обновления создаются для того, чтобы пользователи с помощью встроенной системы обновления могли быстро и просто перейти к использованию новейшей версии Firefox без необходимости загрузки и запуска отдельного установщика. Загрузка пакета обновления происходит быстро и незаметно для пользователя в фоновом режиме. Только после того, как файлы обновления загружены и готовы для применения, Firefox предложит пользователю обновить установленную версию приложения и перезапустить его.

Сложность состоит в том, что мы генерируем *очень много* обновлений. Для серий релизов линии продуктов мы генерируем обновления, которые могут быть применены по отношению ко всем предыдущим поддерживаемым релизам серии для перехода к использованию новейшего релиза для данной линии продуктов. Для Firefox наличие новейшего релиза (`LATEST`) подразумевает необходимость генерации обновлений для каждой платформы, каждой локализации и каждого установщика, начиная с версий Firefox `LATEST-1`, `LATEST-2`, `LATEST-3`, ... в полной и частичной формах. В данный момент мы выпускаем обновления для нескольких различных линий продуктов.

Наша система автоматизации процесса генерации обновлений модифицирует конфигурационные файлы обновления каждой из сборок релиза ветви для поддержания в актуальном состоянии нашего простого списка соответствия между номерами версий, платформами и локализациями, для которых должны быть созданы обновления, позволяющие пользователям перейти на новый релиз. Мы предоставляем обновления в форме "фрагментов" информации. Как вы увидите в примере ниже, этот фрагмент информации является простым файлом в формате XML, расположенным на нашем сервере AUS (службы обновления приложения - Application Update Service), который информирует браузер Firefox на стороне пользователя о том, где расположены полные или частичные архивы обновлений в форме файлов с расширением `.mar` (архив Mozilla - Mozilla Archive).

Важные и незначительные обновления

```
<updates>
<update type="minor" version="7.0.1" extensionVersion="7.0.1"
buildID="20110928134238"
detailsURL="https://www.mozilla.com/en-US/firefox/7.0.1/releasenotes/">
<patch type="complete"
URL="http://download.mozilla.org/?product=firefox-7.0.1-complete&os=osx&lang=en-US&force=1"
hashFunction="SHA512"

hashValue="7ecdbc110468b9b4627299794d793874436353dc36c80151550b08830f9d8c5afd7940c51df9270d54e11fd99806f41368c0f88721
fa17e01ea959144f473f9d"
size="28680122"/>
<patch type="partial"
URL="http://download.mozilla.org/?product=firefox-7.0.1-partial-6.0.2&os=osx&lang=en-US&force=1"
hashFunction="SHA512"

hashValue="e9bb49bee862c7a8000de6508d006edf29778b5dbede4deaf3cfa05c22521fc775da126f5057621960d327615b5186b27d75a378b0
0981394716e93fc5cca11a"
size="10469801"/>
</update>
</updates>
```

Рисунок 2.6: Пример фрагмента информации об обновлении

Как вы можете увидеть на [Рисунке 2.6](#), фрагменты информации об обновлениях содержат атрибут `type`, который может иметь либо значение `major` (важное), либо значение `minor` (незначительное). Незначительные обновления позволяют пользователям перейти к использованию новейшей доступной версии установленного релиза; например, незначительные обновления позволяют всем пользователям релиза 3.6.* перейти к использованию новейшей подверсии для релиза 3.6, всем пользователям бета-версии разрабатываемого релиза - перейти к использованию новейшей бета-версии, всем пользователямочных сборок - перейти к использованию новейшейочной сборки, и.т.д. Наиболее часто выпускаются именно незначительные обновления, которые не требуют от пользователя какого-либо вмешательства в процесс обновления помимо подтверждения намерения выполнения обновления и перезапуска браузера.

Важные обновления используются тогда, когда нам необходимо сообщить пользователям о том, что доступен новейший усовершенствованный релиз, причем в этом случае будет выведено сообщение "Доступна новая версия Firefox, желаете установить обновление?", а также рекламная страница, описывающая наиболее важные функции нового релиза. Внедрение нашей новой системы ускоренного выпуска релизов позволило прекратить выпуск большого количества важных обнов-

лений; у нас появилась возможность прекращения генерации важных обновлений с момента окончания срока поддержки релиза 3.6.*.

Полные и частичные обновления

Во время сборки мы генерируем "полное обновление" в виде файлов с расширением `.tar`, которые содержат все файлы нового релиза, сжатые с помощью компрессора `bz2`, после чего добавленные в файл архива с расширением `.tar`. Как полные, так и частичные обновления автоматически загружаются по каналу доставки обновлений, в котором регистрируется установленная пользователем копия приложения Firefox. Мы используем различные каналы обновлений (то есть, пользователи релизов ожидают обновлений на канале для обновления релизов, пользователи бета-версий ожидают обновлений на канале для обновления бета-версий, и.т.д.), поэтому мы можем доставлять обновления, например, для пользователей релиза и пользователей бета-версии в различное время.

Файлы с расширением `.tar` частичных обновлений создаются путем сравнения файлов с расширением `.tar` для устаревшего релиза с файлами с расширением `.tar` для нового релиза и создания файла "частичного обновления" с расширением `.tar`, содержащего данные различий в бинарной форме для любых измененных файлов, а также файл манифеста. Как вы можете увидеть в примере фрагмента информации об обновлении на [Рисунке 2.6](#) такой подход позволяет значительно сократить размер файлов частичных обновлений. Это очень важно для пользователей, работающих с медленными соединениями с Интернет или с соединениями с Интернет, реализованными по технологии dial-up.

В устаревших версиях нашей системы автоматизации процесс генерации частичных обновлений для всех локализаций и платформ мог длиться до семи часов для одного релиза, так как выполнялась этапы загрузки файлов полных обновлений с расширением `.tar`, поиска различий и упаковки данных частичных обновлений в файл с расширением `.tar`. В конечном счете было установлено, что даже для различных платформ многие изменения компонентов выполнялись идентично, следовательно многие данные различий могли быть повторно использованы. С помощью сценария, который кэшировал хэши для каждой части данных различий, время выполнения нашего процесса генерации обновлений сократилось примерно на 40 минут.

После того, как фрагменты информации об обновлениях загружаются и размещаются на сервере автоматической системы обновлений, шаг проверки наличия обновлений заключается в а) тестовой загрузке фрагментов информации об обновлениях и б) запуске системы обновления приложения для полученного файла с расширением `.tar` для подтверждения корректности применения обновлений.

Генерация файлов частичных обновлений с расширением `.tar`, как и фрагментов информации об обновлениях на данный момент начинается после завершения процесса добавления электронных подписей. Мы выполняем действия в такой последовательности из-за того, что генерация частичных обновлений должна выполняться для файлов двух подписанных релизов и, следовательно, генерация фрагментов информации об обновлениях должна быть задержана до момента доступности подписанных сборок. Как только у нас появится возможность интеграции процесса добавления электронных подписей в процесс генерации сборок, мы сможем генерировать частичные обновления непосредственно после завершения процесса генерации сборки или повторной упаковки. Вместе с усовершенствованиями программного обеспечения нашей системы автоматического обновления, мы получим возможность размещать на зеркалах законченные сборки и повторно упакованные сборки непосредственно после завершения процесса генерации сборок. Это обстоятельство позволяет эффективно осуществлять создание обновлений в параллельном режиме, сокращая общее время их создания на несколько часов.

2.7. Размещение сборок на внутренних зеркалах и контроль качества

Проверка того, что в результате выполнения процесса подготовки релиза был получен ожидаемый результат, является ключевым шагом. Этот шаг осуществляется группой контроля качества в ходе процесса проверки и подтверждения работоспособности сборки.

Как только подписанные сборки становятся доступны, группа контроля качества начинает ручное и автоматизированное тестирование. Контроль качества осуществляется участниками сообщества, нанятыми сторонними специалистами, а также работниками организации, находящимися в различных часовых поясах с целью возможного ускорения этого процесса проверки работоспособности. В то же время наша автоматизированная система подготовки релизов генерирует обновления для всех языков и всех платформ, которые смогут быть применены ко всем поддерживаемым релизам. Фрагменты информации об обновлениях обычно становятся доступны до того момента, как группа контроля качества завершает проверку подписанных сборок. После этого группа контроля качества проверяет возможность осуществления пользователями безопасного обновления при использовании предыдущих релизов для перехода к использованию новейшего релиза с помощью представленных обновлений.

Технически наша система автоматизации перемещает бинарные файлы на наши "внутренние зеркала" (серверы, обслуживаемые организацией Mozilla) с целью проверки обновлений группой контроля качества. Только после того, как группа контроля качества закончит проверку сборок и обновлений, мы перемещаем их на наши зеркала сообщества. Эти зеркала сообщества важны для обработки нагрузки, создаваемой пользователями со всего мира, так как они позволяют пользователям осуществлять запросы обновлений с локальных узлов, на которых размещаются зеркала, вместо отправки запроса напрямую серверу `ftp.mozilla.org`. В том, что мы не делаем сборки и обновления доступными на серверах сообщества до подтверждения их работоспособности группой контроля качества нет ничего плохого, так как в последний момент вполне могут возникнуть сложности в случае обнаружения группой контроля качества серьезной ошибки, в результате чего релиз-кандидат будет отозван.

Процесс проверки работоспособности после завершения генерации сборок и обновлений состоит из следующих этапов:

- Группа контроля качества вместе с участниками сообщества и сторонними специалистами в других часовых поясах проводит тестирование вручную.
- Группа контроля качества запускает автоматизированные системы для проведения функционального тестирования.
- Группа контроля качества независимо проверяет, исправлены ли известные проблемы и достаточно ли высоко качество реализации новых функций для предоставления их в распоряжение пользователей в рамках релиза.
- В это же время система автоматизации процесса подготовки релиза генерирует обновления.
- Группа контроля качества подтверждает работоспособность сборок.
- Группа контроля качества подтверждает работоспособность обновлений.

Следует отметить, что пользователи не получат обновления до тех пор, пока группа контроля качества не подтвердит их работоспособность и координатор релиза не отправит электронное сообщение с командой для перемещения сборок и обновлений в публичный репозиторий.

2.8. Перемещение файлов на публичные зеркала и в систему автоматического обновления

Как только координатор релиза получает подтверждение работоспособности программных компонентов от группы контроля качества и других различных групп организации Mozilla, группа подготовки релизов может продолжить работу, перемещая файлы на серверы из сети зеркал сообще-

ства. Мы используем серверы сообщества для того, чтобы иметь возможность обрабатывать запросы от нескольких сотен миллионов пользователей, загружающих обновления в течение нескольких следующих дней. Все установщики наряду с полными и частичными обновлениями для всех платформ и локализаций в этот момент уже находятся на серверах нашей внутренней сети зеркал. Процесс публикации файлов на внешних зеркалах предполагает внесение изменений в файл исключений приложения rsync для задействования модуля публичных зеркал. После осуществления этих изменений файла начнется синхронизация зеркал, в ходе которой будет осуществлено копирование файлов новых релизов. Каждое зеркало имеет ассоциированный параметр рейтинга или веса; мы отслеживаем то, какие зеркала содержат синхронизированные файлы и суммируем их индивидуальные рейтинги для вычисления общего показателя "распространения" данных. После того, как достигается определенное пороговое значение распространения данных, мы информируем координатора релиза о том, что на зеркалах содержится достаточное количество копий данных для распространения релиза.

Это тот момент, когда релиз становится "официальным". После того, как координатор релиза отправляет финальное сообщение с командой "отправка на публикацию", группа подготовки релизов обновит символные ссылки на веб-сервере, таким образом посетители наших веб- и ftp-ресурсов смогут найти новейшую версию Firefox. Мы также публикуем все фрагменты информации об обновлениях для пользователей прошлых версий Firefox на ресурсах системы автоматического обновления.

Установленные на пользовательских машинах копии приложения Firefox регулярно проверяют доступность обновленной версии Firefox на серверах системы автоматического обновления. После того, как мы публикуем эти фрагменты информации об обновлениях, у пользователей появляется возможность автоматически обновить Firefox до новейшей версии.

2.9. Выученные уроки

Как у инженеров, у нас возникает желание немедленно начать работу над исправлением очевидных технических проблем при их обнаружении. Однако, процесс подготовки релиза затрагивает несколько областей, как технических, так и не технических, поэтому очень важно быть готовым к решению и технических, и не относящихся к техническим вопросов.

Важность взаимодействия с другими заинтересованными сторонами

Важно быть уверенным, что все заинтересованные стороны понимают то, что наш хрупкий процесс подготовки релизов подвергает организацию, а также наших пользователей различным рискам. Эти риски охватывают все уровни организации, допуская потери бизнес-возможностей и изменение позиций на рынке программных продуктов, причиной которых может оказаться медленная и неустойчивая автоматизированная система подготовки релизов. Более того, возможность организации Mozilla защищать своих пользователей путем сверхбыстрого выпуска релизов приобрела большую важность по мере роста пользовательской базы, что в свою очередь сделало наш продукт более привлекательной целью для злоумышленников.

Интересным фактом можно назвать то, что некоторые люди, однажды столкнувшись с неустойчивой системой подготовки релизов во время своей профессиональной деятельности, приходят в организацию Mozilla с негативным отношением к подобным системам, выражаясь утверждением: "да уж, эти системы всегда работают плохо". Объяснение преимуществ, ожидаемых занимающейся бизнесом организацией при внедрении надежной масштабируемой системы автоматизированной подготовки релизов помогает любому человеку понять важность "незаметной" работы группы подготовки релизов, заключающейся в улучшении этой системы, которую мы и пытаемся выполнять.

Привлечение других групп

Для того, чтобы сделать процесс подготовки релизов более эффективным и надежным, группа подготовки релизов, а также другие группы организации Mozilla должны выполнять соответствующую работу. Однако, интересно посмотреть на то, как часто выражение "требуется много времени для выпуска релиза" неверно трактуется, как "группе подготовки релизов требуется много времени для подготовки релиза". Эта неверная трактовка игнорировала работу по подготовке релиза, выполняемую всеми группами за исключением группы подготовки релизов, и снижала мотивацию участников группы подготовки релизов. Исправление этой неверной трактовки требовало дополнительного обучения персонала организации Mozilla, заключающегося в объяснении того, на что на самом деле тратится большая часть времени различными группами в процессе подготовки релиза. Мы проводили это обучение, пользуясь не технологичными "обычными" метками времени в сообщениях электронной почты с четко описанными командами взаимодействия между группами, а также сериями "периодических" записей в блогах с детальным описанием того, на что было потрачено время.

- Это помогло повысить осведомленность людей о том, чем различные группы на самом деле занимаются в процессе подготовки релиза.
- Это помогло людям понять, в каком случае у группы подготовки релизов есть возможность выполнить процесс быстрее, что в свою очередь мотивировало инженеров из этой группы на дальнейшее усовершенствование системы.
- Это помогло участникам других групп задуматься над тем, как они тоже могли бы улучшить процесс подготовки релизов, что было большим изменением мыслительного процесса в рамках всей организации.
- Наконец, это также позволило избавиться от непонятных команд, передаваемых между группами, которые в течение долгого времени приводили к большому количеству ненужных ожиданий, незапланированных стартов системы, а также ко многим другим серьезным нарушениям процесса подготовки релиза.

Установление четких взаимосвязей

Многие из наших "проблем при подготовке релиза" на самом деле являются проблемами людей: нарушение взаимодействия между группами; недостаточно развитые лидерские качества; а также преобладающее стрессовое состояние, утомление и тревога в процессе подготовки chemspill-релизов. После установления четких взаимосвязей между группами для устранения этих нарушений взаимодействий между людьми, процесс подготовки наших релизов сразу же стал более плавным и взаимодействия людей из разных групп очень быстро улучшились.

Управление процессом работы

На начальных этапах реализации данного проекта мы очень часто теряли участников групп. Это само по себе очень плохо. В этой ситуации следует учитывать факт, заключающийся в отсутствии точной актуальной документации и подразумевающий то, что большая часть технических особенностей процесса подготовки релиза была документирована исключительно в устной форме, при этом мы теряли часть данной информации каждый раз, когда участник покидал группу. Нам нужно было срочно изменить сложившееся положение вещей.

Мы пришли к выводу о том, что лучшим способом повышения самосознания и демонстрации улучшения ситуации является предоставление возможности людям убедиться в том, что у нас есть план улучшения ситуации, причем эти люди в некоторой степени смогут изменять его по своему усмотрению. Мы добились этого, гарантировав резервирование времени для исправления по крайней мере одной любой! недоработки после выпуска каждого релиза. Нам удалось реализовать эту возможность путем введения непосредственно после выпуска релиза периода, длящегося в течение одного или двух дней, когда участников групп "не должны беспокоить". Незамедлительное решение мелких проблем в момент, пока о них еще помнят люди, помогло кратковременно отвлечь внимание людей для того, чтобы они впоследствии сфокусировали его на более масштабных проблемах последующих релизов. Более важным является тот факт, что данный подход дал людям ощущение контроля над своим будущим, и процесс подготовки релизов на самом деле начал выполняться более успешно.

Управление изменениями

Из-за давления рынка браузеров бизнес-модель и особенности продуктов организации Mozilla потребовали изменения процесса выпуска релизов в момент нашей работы над его улучшением. Это нестандартная ситуация, которую нужно ожидать.

Мы знали о том, что нам придется выпускать релизы с использованием действующего процесса их подготовки в то время, как мы будем внедрять новый процесс. В связи с этим нами было принято решение об отказе от создания отдельного проекта "greenfield" в процессе работы над существующими системами; мы чувствовали, что действующие системы были настолько хрупкими, что у нас буквально не останется времени на создание чего-либо другого.

Также мы с самого начала предполагали, что нам не будут полностью понятны причины некорректной работы систем. Каждое последовательное улучшение позволяло нам сделать шаг назад и проверить, не таит ли оно в себе новых сюрпризов, перед тем, как начинать работу над новым улучшением. Такие фразы, как "осушение болот", "очистка лука" и "как это вообще работало?" звучали постоянно при обнаружении сюрпризов в ходе работы над данным проектом.

Принимая все это во внимание, мы решили постоянно вносить большое количество незначительных улучшений в существующий процесс. Каждое последовательное улучшение делало следующий релиз немного качественнее. И, что более важно, каждое улучшение высвобождало немного больше времени при выпуске следующего релиза, что позволяло инженеру использовать немного больше времени для внесения следующего улучшения. Эти улучшения накапливались до достижения нами переломного момента, после чего мы начали выделять время для работы над значительными важными улучшениями. В этот момент преимущества оптимизаций процесса подготовки релизов стали действительно заметны.

2.10. Дополнительная информация

Мы действительно гордимся выполненной работой и теми возможностями, которые стали доступны организации Mozilla на новом развивающимся глобальном рынке веб-браузеров.

Четыре года назад выпуск двух chemspill-релизов в месяц был бы темой оживленного обсуждения в Mozilla. В отличие от этого, на прошлой неделе был опубликован экспloit для сторонней библиотеки, из-за чего организации Mozilla пришлось выпустить восемь chemspill-релизов в течение двух неполных рабочих дней.

Принимая во внимание описанные выше возможности, наша система автоматизированной подготовки релизов все еще может быть усовершенствована, при этом наши требования и запросы продолжают меняться. Для того, чтобы узнать о нашей предстоящей работе, обратитесь к следующим источникам информации:

- [Блог Chris AtLee](#)
- [Блог Lukas Blakk](#)
- [Блог John O'Duin](#)
- [Блог Armen Zambrano Gasparnian](#)
- [Документация](#), описывающая архитектуру и принцип выполнения нашего процесса подготовки релизов с использованием системы контроля версий Mercurial
- [Сборочные репозитории](#) группы подготовки релизов. В особенности при подготовке релиза используются репозитории buildbotcustom, buildbot-configs, а также репозитории с кодом вспомогательных инструментов.
- [Документ Firefox 7.0 Beta 4 Build Notes](#). В дополнение к коду мы документируем каждый аспект подготовки релиза. Эта ссылка ведет на документацию для версии 7.0b4, но вы можете перейти к документации для всех остальных релизов в том случае, если отредактируете строку URL соответствующим образом.

3.ОС реального времени FreeRTOS

Глава 3 из книги ["Архитектура приложений с открытым исходным кодом"](#), том 2.

FreeRTOS (произносится как «фри-ар-тосс») является операционной системой реального времени (RTOS) с открытым исходным, которая предназначена для встраиваемых систем. В системе FreeRTOS поддерживается множество различных архитектур и инструментальных средств компиляции, она создавалась как «небольшая, простая и удобная в использовании» система.

Система FreeRTOS находится в стадии активной разработки, которая была начата Ричардом Барри (Richard Barry) в 2002 году. Что касается меня, то я не разработчик системы FreeRTOS или сделал ничего для ее развития, я лишь пользователь и фанат этой системы. В результате, в этой главе, если ее сравнивать с другими главами этой книги, будет больше рассказано о том, «что» и «как» устроено в архитектуре FreeRTOS с меньшим количеством «почему».

Как и во всех операционных системах, основная работа системы FreeRTOS состоит в выполнении задач. Большая часть кода системы FreeRTOS связана с приоритетами, планированием и запуском пользовательских задач. В отличие от других операционных систем, система FreeRTOS является операционной системой реального времени, которая работает во встроенных системах.

Я надеюсь, что к концу этой главы вы поймете базовую архитектуру системы FreeRTOS. Большая часть системы FreeRTOS предназначена для запуска задач, так что вы подробно разберетесь с тем, как именно система FreeRTOS это делает.

Если вы первый раз заглядываете под капот операционной системы, я также надеюсь, что вы усвоите основы того, как работает любая ОС. Система FreeRTOS является относительно простой, особенно в сравнении с Windows, Linux или OS X, но во всех операционных системах используются одни и те же основные концепции и решаются задачи, поэтому может быть поучительно и интересно рассматривать любую операционную систему.

3.1. Что такое «встроенная» и «реального времени»?

«Встроенная» и «реального времени» может означать разное для разных людей, поэтому давайте определим, как эти понятия используются в системе FreeRTOS.

Встроенная система представляет собой компьютерную систему, которая разработана для выполнения всего лишь нескольких задач, например, система в пульте дистанционного управления телевизором, автомобильная система GPS, цифровые часы или кардиостимулятор. Встроенные системы, как правило, меньше и медленнее, чем компьютерные системы общего назначения, и, как правило, также дешевле. Типичная недорогая система может иметь 8-разрядный процессор с тактовой частотой 25 МГц, несколькими килобайтами оперативной памяти, и, возможно, 32 Кб флэш-памяти. В более дорогих встроенных системах может быть 32-разрядный процессор с тактовой частотой 750 МГц, 1 ГБ оперативной памяти, а также несколько гигабайтов флеш-памяти.

Системы реального времени конструируются таким образом, чтобы они успевали что-то выполнить в течение определенного промежутка времени; они гарантируют, что что-то будет выполнено так, как предполагается.

Отличным примером встроенной системы реального времени является кардиостимулятор. Кардиостимулятор должен сокращать сердечную мышцу в нужное время с тем, чтобы мы оставались в живых; он не должен быть слишком занят, чтобы не ответить вовремя. Кардиостимуляторы и другие встроенные системы реального времени разрабатываются очень тщательно с тем, чтобы каждый раз вовремя выполнять свои задачи.

3.2. Обзор архитектуры

Система FreeRTOS является относительно небольшим приложением. Минимальный вариант ядра системы FreeRTOS состоит всего лишь из трех файлов исходного кода (.c) и горстка файлов заголовков, общий размер которых составляет чуть менее 9000 строк кода, включая комментарии и пустые строки. Размер типичного образа системы в исполняемом коде меньше 10 КБ.

Код FreeRTOS подразделяется на три основные части: выполнение задач, коммуникация и интерфейс с аппаратным обеспечением.

- **Задачи:** Почти половина кода ядра FreeRTOS связана с центральной проблемой, решаемой во многих операционных системах: управление задачами. Задачей является определяемая пользователем функция на языке С с заданным приоритетом. В `tasks.c` и `task.h` делается вся тяжелая работа по созданию, планированию и обслуживанию задач.
- **Коммуникации:** Задачи хороши, но задачи, которые могут общаться друг с другом, еще лучше! Это подводит нас ко второй части работ, выполняемых FreeRTOS: коммуникации. Около 40% кода ядра системы FreeRTOS связано с коммуникациями. Обработка коммуникаций в системе FreeRTOS осуществляется в файлах `queue.c` и `queue.h`. Задачи и прерывания используют очереди для передачи данных друг другу, а для того, чтобы сигнализировать об использовании критических ресурсов, применяются семафоры (semaphore) и мютексы (mutex).
- **Подключение к оборудованию:** Около 9000 строк кода, которые составляют базу системы FreeRTOS, являются аппаратно-независимыми; один и тот же код выполняется независимо от того, работает ли система на скромной процессоре 8051 или на новейшем сияющим ядре ARM. Около 6% код ядра системы FreeRTOS представляют собой прослойку между аппаратно-независимым ядром системы FreeRTOS и аппаратно-зависимым кодом. Мы обсудим аппаратно- зависимый код в следующем разделе.

Об аппаратном обеспечении

Аппаратно-независимый слой системы FreeRTOS находится поверх аппаратно-зависимого слоя. В этом аппаратно-зависимом слое известно, как взаимодействовать с архитектурой, построенной на любом чипе, который вы выберете. На рис.3.1 показаны слои системы FreeRTOS.

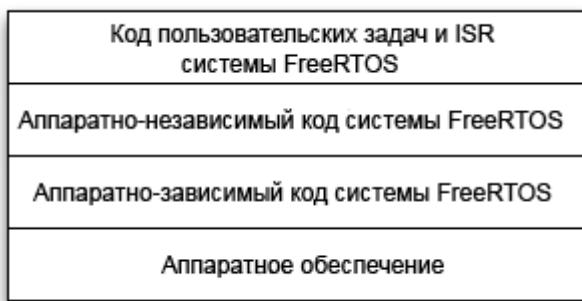


Рис.3.1: Слои программного обеспечения системы FreeRTOS

Система FreeRTOS поставляемый со всем аппаратно-независимым, а также с аппаратно- зависимым кодом, который вы должны получить для того, чтобы настроить и запустить систему. Поддерживается большое количество компиляторов (CodeWarrior, GCC, IAR и т.д.), а также большое количество архитектур процессоров (ARM7, ARM Cortex-M3, различные PIC, Silicon Labs 8051, x86 и т.д.). Список поддерживаемых архитектур и компиляторов смотрите на сайте системы FreeRTOS.

Дизайн системы FreeRTOS позволяет ее легко настраивать в широких пределах. Система FreeRTOS может собрана как для одного процессора, для выполнения только основных функций и

поддержки только нескольких задач, либо она может быть собрана для работы на многоядерном железе с TCP/IP, файловой системой и USB.

Параметры конфигурации выбираются в файле с помощью задания различных значений `#defines`. В этом файле можно выбрать тактовую частоту, размер кучи, мютексы и подмножество API, а также многие другие параметры. Ниже приведено несколько примеров, в которых устанавливается максимальное количество уровней приоритетов задач, частота процессора, тактовая частота системы, минимальный размер стека и общий размер кучи:

```
#define configMAX_PRIORITIES      ( ( unsigned portBASE_TYPE ) 5 )
#define configCPU_CLOCK_HZ        ( 12000000UL )
#define configTICK_RATE_HZ        ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 100 )
#define configTOTAL_HEAP_SIZE    ( ( size_t ) ( 4 * 1024 ) )
```

Аппаратно-зависимую код для каждого инструментального набора компиляции и для каждой архитектуры процессора располагается в отдельных файлах. Например, если вы работаете с компилятором IAR на чипе ARM Cortex-M3, то аппаратно-зависимый код будет находиться в каталоге FreeRTOS/Source/portable/IAR/ARM_CM3/. В файле `portmacro.h` объявляются все аппаратно-зависимые функции, а в файлах `port.c` и `portasm.s` находится весь фактически используемый аппаратно-зависимый код. Во время компиляции аппаратно-независимые заголовки `#include` файла `portable.h` заменяются файлом `portmacro.h`. Система FreeRTOS вызывает аппаратно-зависимые функции с помощью функций `#define`, объявленных в файле `portmacro.h`.

Давайте посмотрим на пример того, как система FreeRTOS вызывает аппаратно- зависимую функцию. Для того, чтобы войти в критическую секцию кода для предотвращения вытеснения, часто требуется аппаратно-независимый файл `tasks.c`. В различных архитектурах вход в критическую секцию происходит по-разному, и, желательно, чтобы аппаратно-независимый файл `tasks.c` не касался деталей, связанных с аппаратной зависимостью. Поэтому `tasks.c` вызывает глобальный макрос `portENTER_CRITICAL()`, радуясь тому, что не требуется знать как и что на самом деле работает. Предположим, что мы используем компилятор IAR на чипе ARM Cortex-M3, система FreeRTOS будет собираться с заголовками

FreeRTOS/Source/portable/IAR/ARM_CM3/portmacro.h, в который макрос `portENTER_CRITICAL()` определяется следующим образом:

```
#define portENTER_CRITICAL() vPortEnterCritical()
```

На самом деле `vPortEnterCritical()` определяется в файле FreeRTOS/Source/portable/IAR/ARM_CM3/port.c. Файл `port.c` является аппаратно- зависимым и содержит код, который понятен компилятору IAR и чипу Cortex-M3. `vPortEnterCritical()` выполняет переход в критическую секцию, используя знания, касающиеся конкретного оборудования, и возвращает управление аппаратно-независимому файлу `tasks.c`.

В файле `portmacro.h` также определяются базовые типы данных для используемой архитектуры. Ниже приведен пример типов данных для базовых целочисленных переменных, указателей и данные тактовой частоты системы для компилятора IAR на чипе ARM Cortex-M3:

```
#define portBASE_TYPE long           // Тип базовой целочисленной переменной
#define portSTACK_TYPE unsigned long // Указатели на позиции в памяти
typedef unsigned portLONG portTickType; // Тип тактовой частоты системы
```

Такой метод использования типов данных и функций через тонкие слои определений `#defines` может показаться несколько сложным, но он позволяет перекомпилировать систему FreeRTOS для архитектуры с совершенно другой системой, изменив только аппаратно- зависимые файлы. И если вы хотите запустить систему FreeRTOS на архитектуре, которая в настоящее время не поддержи-

вается, вам потребуется реализовать только аппаратно-зависимую функциональность, которая значительно меньше, чем аппаратно-независимая часть системы FreeRTOS.

Как мы уже видели, система FreeRTOS реализует аппаратно- зависимую функциональность с помощью макросов `#define` препроцессора языка C. В системе FreeRTOS определения `#define` также используются большей части аппаратно-независимого кода. Для невстраиваемых приложений такое частое использование определений `#define` является кардинальным недостатком, но во многих небольших встроенных системах накладные расходы на вызов функции не дают преимуществ в сравнении с использованием «реальных» функций.

3.3. Планирование задач: Краткий обзор

Приоритеты задач и список готовности

Каждая задача имеет приоритет, назначенный пользователем, который равен от 0 (самый низкий приоритет) и до значения времени компиляции `configMAX_PRIORITIES-1` (самый высокий приоритет). Например, если `configMAX_PRIORITIES` установлен равным 5, то система FreeRTOS будет использовать 5 уровней приоритета: 0 (самый низкий приоритет), 1, 2, 3, и 4 (наивысший приоритет).

В системе FreeRTOS используется «список готовности» («ready list») для отслеживания всех задач, которые в настоящее время готовы к запуску. В ней список готовности реализуется как массив списков задач наподобие следующего:

```
static xList pxReadyTasksLists[ configMAX_PRIORITIES ] ; /* Задачи, готовые согласно приоритетам. */
```

Список `pxReadyTasksLists[0]` является списком всех готовых задач с приоритетом 0, список `pxReadyTasksLists[1]` является списком всех готовых задач с приоритетом 1 и так далее до списка `pxReadyTasksLists[configMAX_PRIORITIES-1]`.

Тактовая частота системы

Пульсом системы FreeRTOS является ее тактовая частота. Система FreeRTOS настраивается таким образом, чтобы периодически выдавать прерывания. Пользователь может регулировать частоту прерываний, которая обычно находится в диапазоне миллисекунд. Каждый раз, когда возникает прерывание, вызывается функция `vTaskSwitchContext()`. Функция `vTaskSwitchContext()` выбирает задачу с наибольшим приоритетом готовности и помещает ее в переменную `pxCurrentTCB`, например, следующим образом:

```
/* Поиск очереди с наивысшим приоритетом, в которой есть готовые к запуску задачи. */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT( uxTopReadyPriority );
    --uxTopReadyPriority;
}

/* listGET_OWNER_OF_NEXT_ENTRY проходит по списку, поскольку задачи с одинаковым приоритетом получают одинаковое право пользоваться процессорным временем. */
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopReadyPriority ] ) );
```

Перед тем как будет запущен цикл, гарантируется, что значение `uxTopReadyPriority` будет больше или равно приоритету задаче, готовой к запуску и имеющей наивысший приоритет. Цикл `while()` начинается с уровня приоритета `uxTopReadyPriority` идвигается вниз по массиву `pxReadyTasksLists[]` с тем, чтобы самый высокий уровень приоритета с задачами, готовыми к

запуску. Затем функция `listGET_OWNER_OF_NEXT_ENTRY()` забирает следующую готовую к запуску задачу из списка готовых задач с этим уровнем приоритета.

Теперь `pxCurrentTCB` указывает на задачу с наивысшим приоритетом, а наиболее приоритетных задач, а когда функция `vTaskSwitchContext()` вернет управление, аппаратно-зависимый кон начнет выполнение этой задачи.

Эти девять строк кода являются, по настоящему, сердцем системы FreeRTOS. Остальные более 8900 строк системы FreeRTOS существуют лишь для того, чтобы удостовериться, что эти девять строк делают все необходимое, чтобы поддерживать выполнение задачи с наибольшим приоритетом.

На рис.3.2 приведена общая схема того, как выглядит список задач, готовых к выполнению. В этом примере есть три уровня приоритета, с одной задачей на уровне приоритета 0, без задач на уровне приоритета 1 и с тремя задачами на уровне приоритета 2. Эта картина абсолютно точная, но не полная, здесь не хватает нескольких деталей, которые мы добавим позже.

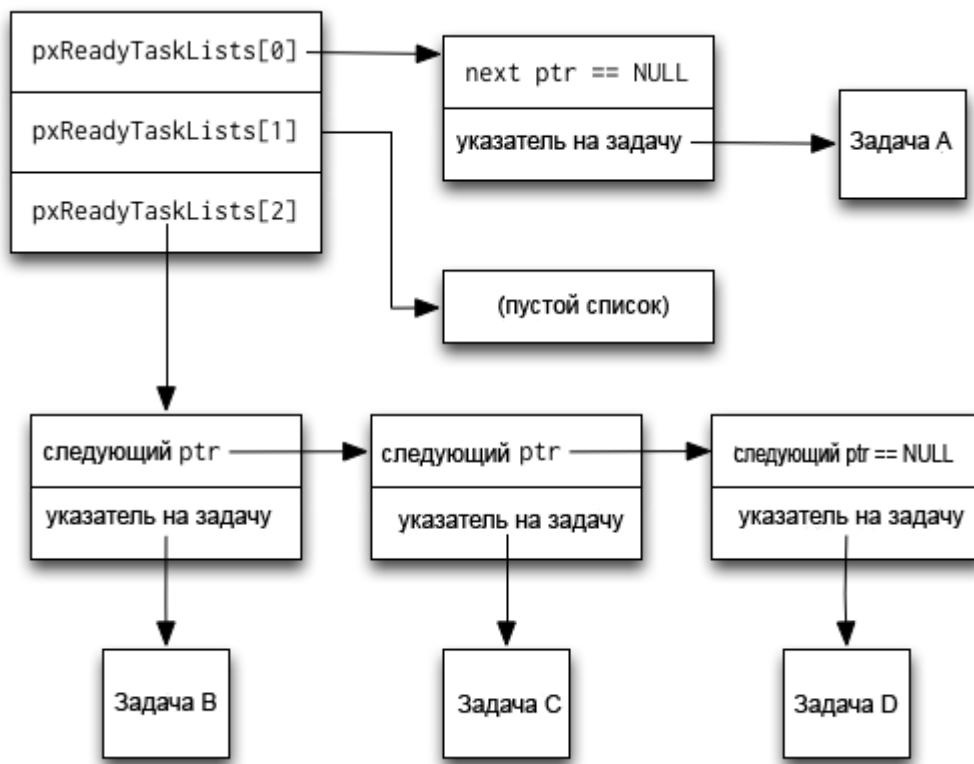


Рис.3.2: Общая схема списка готовности задач Ready List в системе FreeRTOS

Теперь, когда у нас есть общая схема пути, давайте погрузимся в детали. Мы рассмотрим три основные структуры данных системы FreeRTOS: задачи, списки и очереди.

3.4. Задачи

Основная работа всех операционных систем состоит в запуске и координации работы пользовательских задач. Подобно многим операционным системам, основной единицей работы в системе FreeRTOS является задача. В системе FreeRTOS для представления каждой задачи используется блок управления задачей (Task Control Block - TCB).

Блок управления задачей TCB

Блок TCB определяется в tasks.c следующим образом:

```

typedef struct tskTaskControlBlock
{
    volatile portSTACK_TYPE *pxTopOfStack; /* Указывает на местораспо-
ложение
размещенного

МЕНТ

    xListItem     xGenericListItem; /* Элемент списка, использу-
зуемый для
очереди

ных задач. */
    xListItem     xEventListItem; /* Элемент списка, использу-
зуемый для
списки событий.*/
    unsigned portBASE_TYPE uxPriority;

    portSTACK_TYPE *pxStack; /* Приоритет задачи;
0 является низшим
приоритетом. */
    /* Указывает на начало
стека. */
    /* Описательное имя, кото-
присваивается стеку, ко-
создается. Используется
для отладки. */

#if ( portSTACK_GROWTH > 0 )
    portSTACK_TYPE *pxEndOfStack; /* Используется для провер-
ки стека
архитектурах,
где стек растет с млад-
ших
#endif

#if ( configUSE_MUTEXES == 1 )
    unsigned portBASE_TYPE uxBasePriority; /* Приоритет, назначенный
задаче
последним -
используется механизмом
наследования приорите-
тов. */
#endif
} tskTCB;

```

В блоке TCB в переменной pxStack хранится адрес начала стека, а в переменной pxTopOfStack - текущая вершина стека. В нем также в переменной pxEndOfStack хранится указатель на конец стека для проверки стека на переполнение в случае, если стек растет «вверх» в сторону старших адресов. Если стек растет «вниз» к младшим адресам, то переполнение стека проверяется путем сравнения текущей вершины стека с началом стека, которое хранится в переменной pxStack.

В блоке TCB в переменных `uxPriority` и `uxBasePriority` хранится начальный приоритет задачи. Задача дается приоритет, когда она создается, и приоритет задачи может быть изменен. Если в системе FreeRTOS реализовано наследование приоритетов, то переменная `uxBasePriority` используется для вспоминания первоначального приоритета, когда временно возвращается до «наследуемого» приоритета. Подробности, касающиеся наследования приоритетов, приведены ниже в обсуждении мютексов.

В каждой задаче есть два элемента списка для использования в различных списках планирования в системе FreeRTOS. Когда задача добавляется в список, система FreeRTOS не вставляет указатель непосредственно в блок TCB. Вместо этого, он вставляет указатель либо в переменную `xGenericListItem`, либо в переменную `xEventListItem` блока TCB. Эти переменные `xListItem` позволяют системе FreeRTOS организовывать списки более хитро, чем если бы в них был указатель на блок TCB. Мы увидим это на примере позже, когда будем обсуждать списки.

Задача может находиться в одном из четырех состояний: выполняться, готова к выполнению, приостановлена или блокирована. Можно было бы ожидать, что в каждой задаче есть переменная, которая сообщает системе FreeRTOS о том, в каком состоянии задача находится, но это не так. Вместо этого, система FreeRTOS отслеживает состояние задачи неявно, помещая задачи в соответствующий список: список задач, готовых для выполнения, список приостановленных задач и т.д. Присутствие задачи в конкретном списке указывает состояние задачи. Когда задача переходит из одного состояния в другое, система FreeRTOS просто перемещает ее из одного списка в другой.

Настройка задачи

Мы уже затронули вопрос о том как задача выбирается и как планируется на исполнение с массивом `pxReadyTasksLists`; теперь давайте посмотрим на то, как первоначально создается задача. Задача создается, когда вызывается функция `xTaskCreate()`. FreeRTOS использует только что выделенный блок TCB объект для хранения имени, приоритета и другие детали, касающиеся задачи, а затем выделяет некоторое количество памяти из стека по запросам пользователя (если в наличии есть достаточно памяти) и запоминает начало стека в элементе `pxStack` блока пользователя TCB.

Стек инициализируется таким образом, как будто новая задача уже запущена и была прервана переключением контекста. Таким образом, планировщик может рассматривать новые только что созданные задачи точно так же, как те, что уже работали некоторое время; планировщику не требуется какой-либо специальный код для обработки новых задач.

Способ, с помощью которого стек задачи создается таким образом, чтобы он выглядел, как если бы задача была прервана переключением контекста, зависит от архитектуры, на которой работает система FreeRTOS; хорошим примером является следующая реализация для процессора ARM Cortex-M3:

```
unsigned int *pxPortInitialiseStack( unsigned int *pxTopOfStack,
                                    pdTASK_CODE pxCode,
                                    void *pvParameters )
{
    /* Эмулирует фрейм стека как если бы он был создан прерыванием переключателя контекста. */
    pxTopOfStack--; /* Смещение добавляется к значению счетчика — так MCU использует стек при
                      переходе на прерывание/выходе из прерывания. */
    *pxTopOfStack = portINITIAL_XPSR; /* xPSR */
    pxTopOfStack--;
    *pxTopOfStack = ( portSTACK_TYPE ) pxCode; /* PC */
    pxTopOfStack--;
    *pxTopOfStack = 0; /* LR */
    pxTopOfStack -= 5; /* R12, R3, R2 and R1. */
    *pxTopOfStack = ( portSTACK_TYPE ) pvParameters; /* R0 */
```

```

pxTopOfStack -= 8; /* R11, R10, R9, R8, R7, R6, R5 and R4. */
return pxTopOfStack;
}

```

Когда происходит прерывание задачи процессор ARM Cortex-M3 помещает регистры в стек, когда задача прерывается. Функция `pxPortInitialiseStack()` изменяет стек так, чтобы он выглядел как будто в него были помещены регистры, хотя в действительности задача даже не начала выполняться. Для регистров `xPSR`, `PC`, `LR` и `R0` процессора ARM в стеке хранятся известные значения. Остальные регистры `R1 - R12` получить в стеке память, выделенное для них путем уменьшения верхней части указателя стека, но для этих регистров в стеке не хранятся какие-либо конкретные данных. В архитектуре ARM определяется, что при перезагрузке значения этих регистров не определены, поэтому в программе (не имеющей ошибок) не следует считать, что там хранятся известные значения.

После того, как стек подготовлен, задача почти готова к запуску. Однако, во-первых, система FreeRTOS отключает прерывания: Мы собираемся начать с передачей готовых списков и других структур планировщика, и мы не хотим, чтобы кто-нибудь, кроме нас, их менял.

Если это первая задача, которую когда-либо была создана создан, то система FreeRTOS инициализация списки задач планировщика. В планировщике системы FreeRTOS есть массив списков готовых задач `pxReadyTasksLists[]`, в котором для каждого возможного уровня приоритета есть один список готовых задач. В системе FreeRTOS также есть несколько других списков, используемых для отслеживания задач, которые были приостановлены, уничтожены или задержаны. Сейчас они все инициализируются.

После того, как будет сделана любая первая инициализация, новая задача добавляется в список готовых задач, соответствующий указанному в ней уровню приоритета. Будут активированы прерывания и создание новой задачи завершится.

3.5. Списки

После задач, следующими наиболее часто используемыми в системе FreeRTOS являются списки. Система FreeRTOS использует структуру списков для отслеживания состояния задач при планирования, а также для реализации очередей.

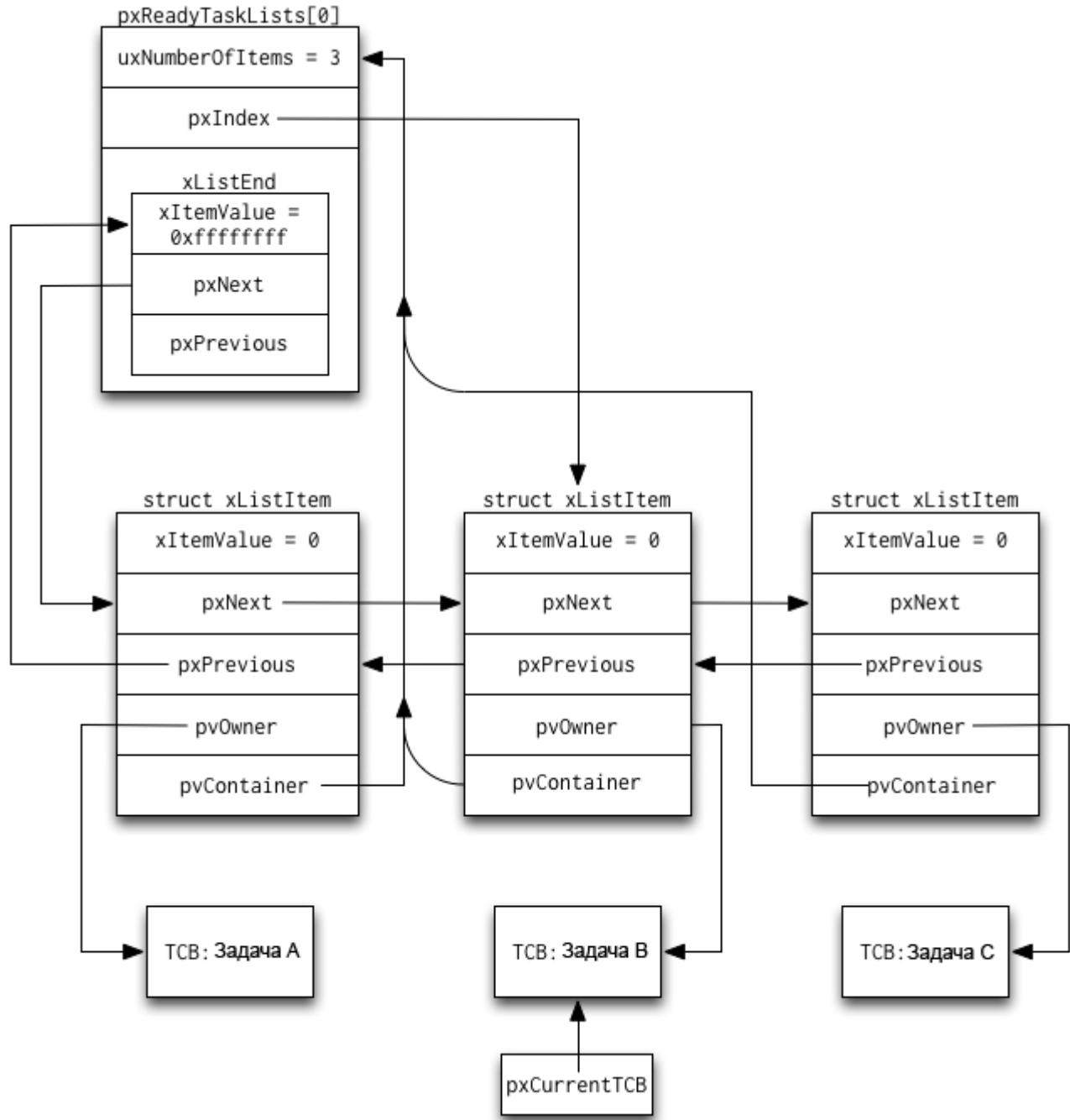


Рис.3.3: Полная схема списка готовности задач Ready List в системе FreeRTOS

Список в системе FreeRTOS является стандартным закольцованным двусвязным списком с парой интересных дополнений. Элементы списка следующие:

```

    void * pvOwner;                                /* Указатель на объект (обычно блок
TCB),                                         в котором находится элемент списка.

Таким                                         образом, организуется двусвязный спи-
сок                                         между объектами, хранящимися в спи-
ске, и                                         элементами самого списка. */
    void * pvContainer;                           /* Указатель на список (если таковой
имеется),                                         в который этот элемент списка помеща-
ется. */                                     ;
};


```

В каждом элементе хранится номер, `xItemValue`, которое обычно является приоритетом задачи, который отслеживается, или значением таймера для планирования событий. Списки хранятся в порядке убывания приоритета, а это означает, что наивысший приоритет `xItemValue` (наибольшее число) находится в начале списка и самый низкий приоритет `xItemValue` (наименьшее число) находится в конце списка.

Указатели `pxNext` и `pxPrevious` являются стандартными указателями, связывающими элементы списка. Указатель `pvOwner` является указателем на владельца элемента списка. Обычно это указатель на блок TCB задачи. Указатель `pvOwner` используется для быстрого переключения задач в `vTaskSwitchContext()`: как только в `pxReadyTasksLists[]` будет найден элемент списка с наивысшим приоритетом, указатель `pvOwner` элемента списка позволит нам непосредственно перейти к блоку TCB, который нужен при планировании запусков задачи.

Указатель `pvContainer` указывает на список, в котором находится этот элемент. Он используется для быстрого определения, принадлежит ли элемент некоторому списку. Каждый элемент списка может быть помещен в список, что осуществляется следующим образом:

```

typedef struct xLIST
{
    volatile unsigned portBASE_TYPE uxNumberOfItems;
    volatile xListItem * pxIndex;                  /* Используется для прохода по списку. Ука-
зывает
                                                 на последний элемент, возвращенный
                                                 функцией pvListGetOwnerOfNextEntry () . */

    volatile xMiniListItem xListEnd;               /* Элемент списка, в котором находится мак-
симально
                                                 возможное значение, означающее, что он
                                                 находится в конце списка и, поэтому,
                                                 используется как маркер. */
} xList;

```

Размер списка в любое время хранится в переменной `uxNumberOfItems`, предназначенней для быстрого выполнения операций с размерами списков. Все новые списки инициализируются таким образом, что в них есть один элемент: элемент `xListEnd`. Значение `xListEnd.xItemValue` является контрольным значением, равным наибольшему значению для переменной `xItemValue`: `0xffff` в случае, если `portTickType` представляет собой 16-битное значение, и `0xffffffff` в случае, если `portTickType` представляет собой 32-битное значение. Другие элементы списка также могут иметь такое же значение; алгоритм вставки элементов списка гарантирует, что `xListEnd` всегда будет последним элементом в списке.

Поскольку списки сортируются в порядке убывания, элемент `xListEnd` используется как маркер начала списка. А поскольку список кольцевой, этот элемент `xListEnd` также является маркером конца списка.

В большинстве «традиционных» операций доступа к списку, которыми вы пользуетесь, вся работа выполняется в одном цикле `for()` или в функции, вызываемой следующим образом:

```
for (listPtr = listStart; listPtr != NULL; listPtr = listPtr->next) {  
    // Что-то здесь делается с указателями listPtr ...  
}
```

В системе FreeRTOS часто требуется получить доступ к спискам сразу с помощью нескольких циклов `for()` и `while()`, а также с помощью нескольких вызовов функций, и поэтому происходит обращение к функциям, в которых при обходе списка используется указатель `pxIndex`. Функция `listGET_OWNER_OF_NEXT_ENTRY()` выполняет операцию `pxIndex = pxIndex->pxNext;` и возвраща-ет значение `pxIndex`. (Конечно, также осуществляется соответствующее определение конца спи-ска). Таким образом, во время перемещения по списку с использованием `pxIndex` сам список от-вечает за отслеживание «текущего положения», позволяя остальной части системы FreeRTOS не беспокоиться об этом.

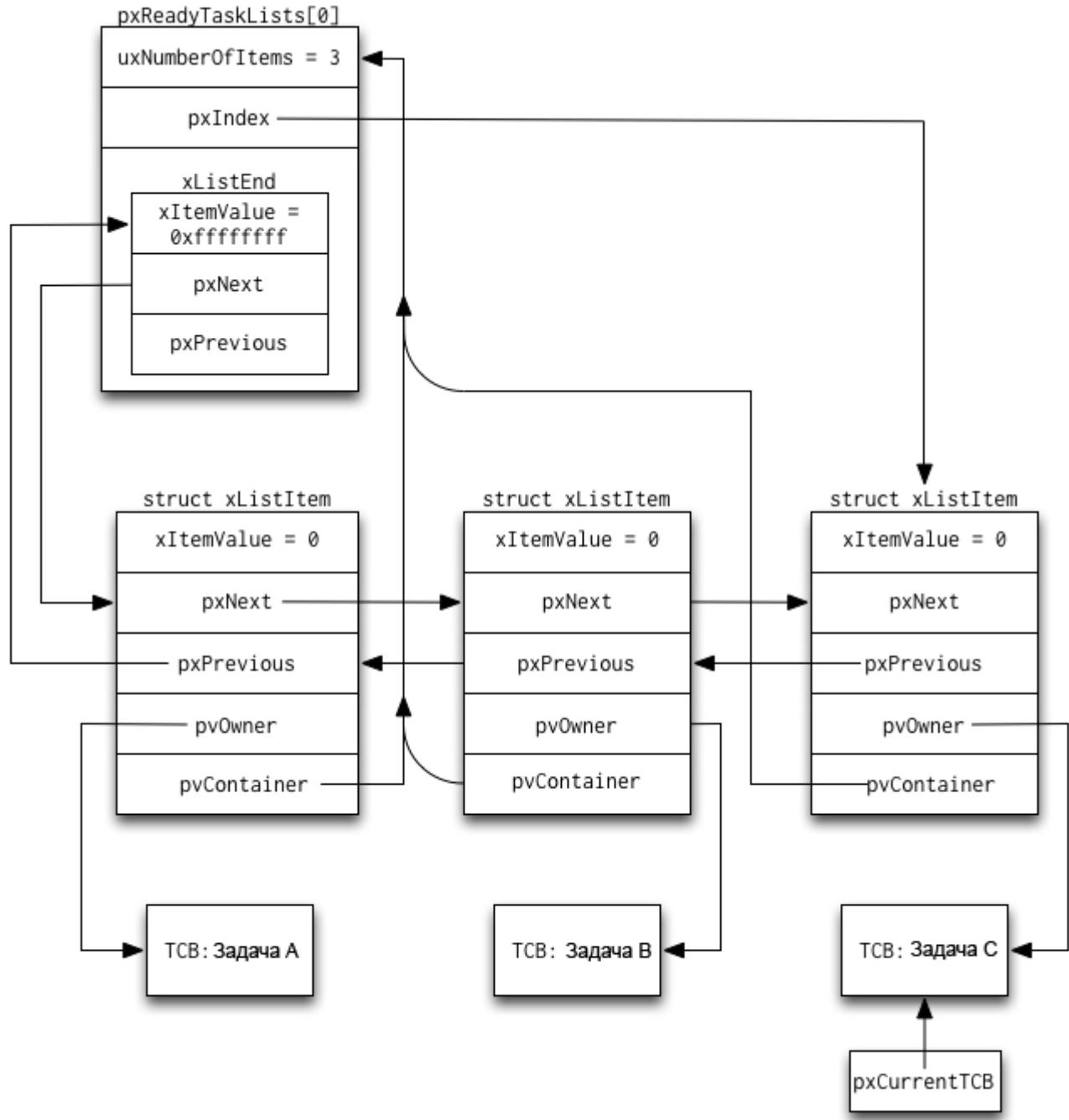


Рис.3.4: Полная схема списка готовности задач Ready List в системе FreeRTOS после возникновения прерывания

То, что все действия со списком `pxReadyTasksLists[]` выполняются в функции `vTaskSwitchContext()`, является хорошим примером того, как используется указатель `pxIndex`. Давайте предположим, что у нас есть только один уровень приоритета, приоритет 0, и есть три задачи на этом уровне приоритета. Это похоже на общую схему списка готовности задач, которую мы рассмотрели ранее, но на этот раз мы будем рассматривать все структуры и поля данных.

Как видно на рис.3.3, `pxCurrentTCB` указывает, что в настоящее время выполняется Задача В. В следующий момент времени выполняется функция `vTaskSwitchContext()`, она вызывает функцию `listGET_OWNER_OF_NEXT_ENTRY()` с тем, чтобы перейти к запуску следующей задачи. Эта функция использует `pxIndex->pxNext` для того, чтобы выяснить, что следующей задачей является Задача С, и теперь `pxIndex` указывает на элемент списка Задачи С, а `pxCurrentTCB` указывает на блок ТСВ Задачи С так, как показано на рис.3.4.

Обратите внимание, что каждый объект struct xListItem является, на самом деле, объектом xGenericListItem из соответствующего блока TCB.

3.6. Очереди

Система FreeRTOS позволяет задачам с помощью очередей общаться и синхронизироваться друг с другом. Процедуры сервиса прерываний (ISR) также используют очереди для взаимодействий и синхронизации.

Базовая структура данных очереди выглядит следующим образом:

```
typedef struct QueueDefinition
{
    signed char *pcHead;                                /* Указывает на начало области хранения
                                                        очереди. */
    signed char *pcTail;                                /* Указывает на байт в конце области хра-
нения
                                                        очереди. Еще один байт требуется по-
скольку
                                                        хранятся отдельные элементы очереди;
он используется как маркер. */
    signed char *pcWriteTo;                            /* Указывает на следующее свободное место
в
                                                        в области хранения очереди. */
    signed char *pcReadFrom;                           /* Указывает на последнюю позицию, откуда
происходило чтение очереди. */

    xList xTasksWaitingToSend;                         /* Список задач, которые блокированы,
ожиная
                                                        пока не произойдет обращение к этой
очереди;
                                                        Запомнены в порядке приоритета. */
    xList xTasksWaitingToReceive;                      /* Список задач, которые блокированы,
ожиная
                                                        пока не произойдет чтение из этой оче-
реди;
                                                        Запомнены в порядке приоритета. */

    volatile unsigned portBASE_TYPE uxMessagesWaiting; /* Количество элементов, имею-
                                                        в очередь в текущий момент.
*/
    unsigned portBASE_TYPE uxLength;                   /* Длина очереди, определяемая
                                                        количество элементов, наход-
ящихся в
                                                        очередь, а не как количе-
ство
                                                        байтов памяти, занимаемой
очередью. */
    unsigned portBASE_TYPE uxItemSize;                /* Размер каждого элемента, ко-
торый
                                                        хранится в очереди. */

} xQUEUE;
```

Это довольно стандартная очередь с указателями начала и конца, а также с указателями, позволяющими отслеживать, откуда мы только что выполнили чтение и куда мы только что сделали запись.

Когда создается очередь, пользователь указывает длину очереди и размер каждого элемента, который будет отслеживаться с помощью очереди. Указатели pcHead и pcTail используются для от-

слеживания того, как очередью используется внутренняя память. При добавлении элемента в очередь делается полная копия элемента во внутреннюю область хранения очереди.

Система FreeRTOS делает полную копию, а не хранит указатель на элемент, поскольку время жизни вставляемого элемента может быть намного короче, чем время жизни очереди. Рассмотрим, например, очередь из простых целых чисел, вставляемых и удаляемых с помощью локальных переменных в нескольких вызовах функций. Если в очереди хранятся указатели на целочисленные локальные переменные, то указатели станут недействительными как только целочисленные локальные переменные пропадут области видимости и память, используемая для локальных переменных, будет использована для некоторого нового значения.

Пользователь выбирает что следует помещать в очередь. Пользователь может помещать в очередь копии самих элементов, если элементы небольшие, например, как простые целые числа из предыдущего абзаца, либо пользователь может помещать в очередь указатели на элементы, если элементы большие. Обратите внимание, что в обоих случаях система FreeRTOS делает полную копию элементов: если пользователь выберет помещать в очередь копии элементов, то в очереди хранится полная копия каждого элемента, если пользователь выбирает помещать в очередь указатели, то в очереди хранится точная копия указателя. Конечно, если пользователь хранит в очереди указатели, то на него возлагается ответственность за управление памятью, связанной с указателями. Очереди безразлично, какие данные в ней хранятся, она просто должна знать размер каждого элемента данных.

В системе FreeRTOS поддерживаются вставки и удаления в очереди с блокировкой и без блокировки. Неблокирующие операции возвращают управление немедленно с указанием состояния «Вставлен элемент в очередь?» или «Удален элемент из очереди?». Блокирование указываются с тайм-аутом. Задача может ожидать снятие блокировки бесконечно или в течение ограниченного периода времени.

Заблокированная задача, назовем ее Задачей А, будет оставаться заблокированной до тех пор, пока не будет завершено выполнение операции вставки/удаления или пока не истечет (если оно установлено) время тайм-аута. Если прерывание или другая задача изменит очередь так, что может быть закончена операция для Задачи А, то Задача А будет разблокирована. Если операция в очереди, выполняемая для Задачи А можно выполнить в течение того, времени, пока задача выполняется, то Задача А завершит свою операцию с очередью и вернет значение состояния «success» (успешное завершение). Однако, в течение того времени, пока Задача А выполняется, может случиться так, что задача с более высоким приоритетом или прерывание выполнить еще одну операцию в очереди, которая помещает Задаче А выполнить свою операцию. В этом случае Задача А проверить значение тайм-аута и, если тайм-аут еще не истек, то продолжит оставаться заблокированной, либо вернет значение «failed» (не выполнено) в качестве состояния выполнения операции.

Важно отметить, что пока задача заблокирована в очереди, остальная часть системы будет продолжать работать; другие задачи и прерывания будут продолжать выполняться. Таким образом, заблокированная задача их не тратит ресурсы процессора, которые могут быть продуктивно использованы другими задачами и прерываниями.

В системе FreeRTOS используется список `xTasksWaitingToSend` для отслеживания задач, которые заблокированы при выполнении операции вставки элемента в очередь. Каждый раз, когда элемент удаляется из очереди, проверяется список `xTasksWaitingToSend`. Если задача находится в состоянии ожидания в этом списке, то задача разблокируется.

Аналогично, с помощью списка `xTasksWaitingToReceive` отслеживаются задачи, которые заблокируются при выполнении операции удаления из очереди. Каждый раз, когда новый элемент вставляется в очередь, проверяется список `xTasksWaitingToReceive`. Если задача находится в состоянии ожидания в этом списке, то задача разблокируется.

Семафоры и мютексы

Система FreeRTOS использует очереди для обмена данными между задачами и внутри задач. Система FreeRTOS также использует очереди для реализации семафоров и мютексов.

В чем разница?

Семафоры и мютексы могут рассматривать почти как одно и то же, но это не так. В системе FreeRTOS они реализуют аналогичным образом, но они предназначены для использования по-разному. Как они могут использоваться по-разному? Гуру встроенных систем Майкл Барр (Michael Barr) лучше всего это описывает в своей статье [«Mutexes and Semaphores Demystified»](#) («Демистификация мютексов и семафоров»):

Правильное использование семафора состоит в передаче сигнала от одной задачи в другую. Смысл мютексов в том, что каждая задача обращается к ним и отказывается от их использования в том порядке, в каком с их помощью устанавливается защита на совместно используемые ресурсы. В отличие от задач, в которых используются семафоры, может быть либо послан некоторый сигнал («send» или «отправить в терминах системы FreeRTOS»), либо может происходить ожидание сигнала («receive» или «получить» в терминах системыFreeRTOS), но не оба варианта.

Мютекс используется для защиты общего ресурса. Задача включает мютекс, использует общий ресурс, а затем отключает мютекс. Никакая задача не может включить мютекс, пока мютекс включен другой задачей. Это гарантирует, что в каждый конкретный момент общим ресурсом может пользоваться только одна задача.

Семафоры, используемые некоторой задачей, посыпают сигнал другой задаче. Процитирую статью Барра:

Например, в Задаче 1 может быть код, который, когда нажата кнопка «power» («Питание»), посылает сообщение (т.е. выдает сигнал или увеличивает на единицу некоторое значение) конкретному семафору, а Задача 2, которая включает дисплей, ожидает сигнала от того же самого семафора. В этом случае одна задача создает сигнал, а другая — его потребляет.

Если вам не все понятно с семафорами и мютексами, пожалуйста, ознакомьтесь со статьей Майкла.

Реализация

В системе FreeRTOS реализован N-элементный семафор в виде очереди, в которой может быть N элементов. В ней не хранятся какие-либо данные в виде элементов очереди; семафор просто следит за тем, сколько записей в текущий момент помещено в очередь, что осуществляется с помощью поля uxMessagesWaiting, имеющегося в очереди. Семафор реализует «чистую синхронизацию» так, как это названо в вызовах заголовочного файла semphr.h системы FreeRTOS. Поэтому размер элемента в очереди указан равным нулю байтов (uxItemSize == 0). Каждое обращение к семафору увеличивает или уменьшает на единицу значение поля uxMessagesWaiting; копирование элементов очереди или данных выполнять не требуется.

Точно также, как и семафоры, мютексы реализованы в виде очередей, но в них с помощью определений #defines перегружены несколько полей xQUEUE:

```
/* Перепределение полей структуры xQUEUE. */
#define uxQueueType          pCHead
#define pxMutexHolder        pCTail
```

Поскольку мютекс не хранит никаких данных в очереди, ему не нужна внутренняя память, и, поэтому, не нужны поля `pcHead` и `pcTail`. Система FreeRTOS устанавливает поле `uxQueueType` (в действительности поле `pcHead`) равным 0, указывая, что эта очередь используется для мютекса. Система FreeRTOS использует перегрузку полей `pcTail` для реализации в мютексах механизма наследования приоритетов.

В случае, если вы не знакомы с наследованием приоритетов, я для того, чтобы определить его, еще раз процитирую Майкла Барра, на этот раз его статью [«Introduction to Priority Inversion»](#) («Введение в инверсии приоритетов»):

[Наследование приоритетов] разрешает чтобы задача с низким приоритетом наследовала приоритет любой задачи более высокого приоритета, ожидающей ресурс, которым эти задачи пользуются совместно. Такое изменение приоритета должно происходить сразу, как только задача с большим приоритетом переходит в состояние ожидания; оно должно прекращаться сразу, как только ресурс будет освобожден.

Система FreeRTOS реализует наследование приоритетов с использованием поля `pxMutexHolder` (которое, на самом деле, является перегруженным полем `#define pcTail`). Система FreeRTOS записывает задачу, которая использует мютекс, в поле `pxMutexHolder`. Когда будет обнаружено, что задача с более высоким приоритетом также пользуется мютексом, установленным задачей с низким приоритетом, система FreeRTOS «обновит» приоритет задачи с более низким приоритетом до приоритета задачи с более высоким приоритетом и будет его поддерживать таким до тех пор, пока первая задача не освободит ресурс.

3.7. Заключение

Мы завершили обзор архитектуры FreeRTOS. Надеюсь, теперь вы понимаете, как в системе FreeRTOS выполняются задачи и как между ними происходит взаимодействие. И если вы раньше никогда не заглядывали внутрь какой-нибудь ОС, то, я надеюсь, что теперь у вас есть общее представление о том, как они работают.

Очевидно, что в этой главе не рассмотрены все особенности архитектуры системы FreeRTOS. Следует отметить, что я не упомянул о выделении памяти, системе прерываний ISR, средствах отладки и о поддержке MPU. В этой главе также не обсуждалось, как настраивать и использовать систему FreeRTOS. Ричард Барри (Richard Barry) написал отличную книгу, [Using the FreeRTOS Real Time Kernel: A Practical Guide](#) («Использование ядра реального времени FreeRTOS: Практическое руководство»), в которой обсуждается именно это, я настоятельно ее рекомендую, если вы собираетесь пользоваться системой FreeRTOS.

3.8. Благодарности

Я хотел бы поблагодарить Ричарда Барри (Richard Barry) для создания и сопровождение системы FreeRTOS, а также за то, что он решил ее сделать проектом с открытым исходным кодом. Ричард очень помог при написании данной главы, рассказав немного об истории системы FreeRTOS, а также сделав замечания, очень ценные с технической точки зрения.

Спасибо также Эми Браун (Amy Brown) и Грэгу Уилсону (Greg Wilson) за то, что они втянули меня в проект AOSA.

Последняя по упоминанию и наиболее важная благодарность моей супруге Саре за совместное участие в исследованиях и написании этой главы. К счастью, она знала, что я гик, когда выходила за меня замуж!

4.GDB

Глава 4 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

GDB или отладчик GNU был одной из первых программ, разработанных для нужд Фонда свободного программного обеспечения и ставших важнейшим элементом экосистемы распространяемого под свободными лицензиями программного обеспечения с открытым исходным кодом с момента разработки. Изначально архитектура приложения подразумевала его использование в роли обычного отладчика уровня исходного кода для Unix, но с тех пор она подверглась расширению для добавления широкого спектра вариантов использования, включая использование отладчика во многих встраиваемых системах, причем кодовая база приложения увеличилась с нескольких тысяч до полумиллиона строк на языке C.

В данной главе мы в общих чертах рассмотрим внутреннюю структуру отладчика GDB с указанием на стадии поэтапного процесса разработки, ознаменовывающие появление у пользователей требований в отношении новых возможностей приложения с течением времени.

4.1. Цель разработки

Приложение GDB было спроектировано для использования в роли символьного отладчика для программ, разработанных с использованием таких компилируемых императивных языков программирования, как C, C++, Ada и Fortran. При использовании оригинального интерфейса командной строки приложения будет получен вывод, подобный следующему:

```
% gdb myprog
[...]
(gdb) break buggy_function
Breakpoint 1 at 0x12345678: file myprog.c, line 232.
(gdb) run 45 92
Starting program: myprog
Breakpoint 1, buggy_function (arg1=45, arg2=92) at myprog.c:232
232      result = positive_variable * arg1 + arg2;
(gdb) print positive_variable
$1 = -34
(gdb)
```

GDB сообщает о том, что с отлаживаемым приложением что-то не так, разработчик говорит "ага" или "тммм", после чего ему предстоит дать ответы на вопросы о том, в чем заключается ошибка и как ее исправить.

Важная особенность архитектуры заключается в том, что подобный GDB инструмент при обобщенном рассмотрении является набором интерактивных инструментов для углубленного исследования программы и, следовательно, он должен иметь возможность ответить на серии непредсказуемых запросов. В дополнение к этому он будет использоваться для отладки оптимизированных средствами компилятора программ, а также программ, использующих каждую доступную аппаратную возможность для увеличения производительности, поэтому у него должна быть необходимая информация и возможность полноценной работы на низких системных уровнях.

У GDB также должна быть возможность отладки программ, скомпилированных с помощью сторонних компиляторов (не только компилятора GNU C), программ, скомпилированных давным-давно с помощью устаревших версий компиляторов, а также программ, информация о символах которых отсутствует, устарела или просто является некорректной; следовательно, еще одно архитектурное решение должно заключаться в том, что отладчик GDB должен продолжать работать и

успешно выполнять поставленные перед ним задачи даже в том случае, если данные, относящиеся к внутреннему устройству приложения, отсутствуют, повреждены или просто некорректны.

В последующих разделах будут рассматриваться вопросы, подразумевающие наличие у читателя опыта использования интерфейса командной строки GDB. Если вы практически не знакомы с GDB, займитесь изучением руководства. [[SPS+00](#)].

4.2. Начало развития GDB

GDB является довольно старой программой. Он начал свое существование примерно в 1985 году как разработка от Richard Stollman, поставляемая вместе с GCC, GNU Emacs и другими ранними версиями компонентов проекта GNU. (В это время не существовало публичных репозиториев систем контроля версий, поэтому большая часть подробностей истории разработки на данный момент потеряна.)

При сравнении исходного кода ранних релизов от 1988 года с исходным кодом современных релизов можно найти всего лишь несколько похожих строк; практически весь исходный код GDB был переработан по крайней мере единожды. Другая заметная особенность ранних версий GDB заключается в достаточно скромных начальных целях разработки проекта, поэтому большая часть работы с того момента заключалась в расширении возможностей приложения GDB для работы в различных окружениях и расширении вариантов его использования, что не предусматривалось изначальным планом разработки.

4.3. Блочная диаграмма

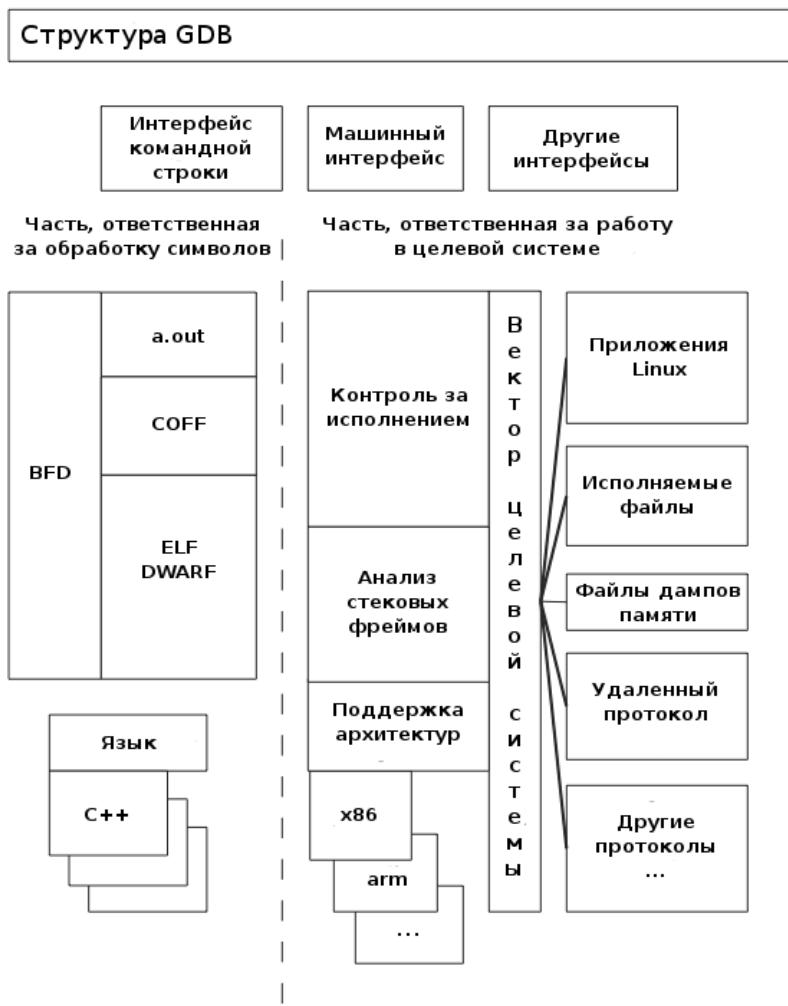


Рисунок 4.1: Обобщенная структура GDB

При отдаленном рассмотрении можно сделать вывод о том, что приложение GDB состоит из двух частей:

1. Часть, ответственная за обработку символов, занимается работой с информацией о символах программы. Информация о символах включает имена функций и переменных, информацию о типах данных, номера строк, информацию об использовании машинных регистров и другую подобную информацию. В части, ответственной за обработку символов, происходит извлечение этой информации из исполняемого файла программы, разбор полученных выражений, поиск адресов в памяти для заданного номера строки, поиск в исходном коде и общая работа с программой по мере написания ее исходного кода разработчиком.
2. Часть, ответственная за работу в целевой системе, занимается непосредственным взаимодействием с целевой системой. Она позволяет запустить и остановить исполнение программы, считать данные из памяти и регистров, изменить их, перехватить сигналы, а также выполнить другие аналогичные действия. Специфика реализации этих операций значительно отличается для различных систем; большинство Unix-подобных систем предоставляет специальный системный вызов с именем `ptrace`, который дает возможность одному процессу читать и записывать данные состояния другого процесса. Таким образом, часть GDB, ответственная за работу в целевой системе, в большинстве случаев использует системный вызов `ptrace` и интерпретирует получаемые результаты. Однако, в случае кросс-отладки встраиваемой системы, часть, ответственная за работу в целевой системе, формирует пакеты сообщений для отправки по сети и ожидает пакетов, отправленных в ответ.

Эти две части в каком-то смысле независимы друг от друга; вы можете исследовать код вашей программы, просматривать типы переменных, и.т.д., без непосредственного запуска программы. С другой стороны, в также можете заниматься отладкой необработанного машинного кода даже в том случае, если информация о символах приложения не доступна.

Объединяющим две рассмотренные части и находящимся посередине звеном является командный интерпретатор и главный управляющий цикл исполнения.

4.4. Примеры работы

В качестве простого примера взаимодействия описанных частей рассмотрим команду `print`, описанную выше. Командный интерпретатор ищет функцию, соответствующую команде `print` и предназначенную для разбора выражения и преобразования его в простую древовидную структуру, которая в последствии подвергнется обработке путем обхода дерева. В какой-то момент будет осуществлено обращение к таблице символов для установления того, что `positive_variable` является целочисленной глобальной переменной, которая хранится, скажем, по адресу `0x601028` в памяти. После этого осуществляется вызов функции из части, ответственной за работу в целевой системе, для чтения четырех байт из памяти по указанному адресу с последующей передачей этих байт функции форматированного вывода, которая выведет их в формате числа в десятичной системе счисления.

Для вывода исходного кода и его скомпилированной версии GDB осуществляет комбинацию операций чтения данных из файла исходного кода и целевой системы, после чего использует сгенерированную компилятором информацию о номере строки для объединения двух представлений. В приведенном здесь примере строка 232 имеет адрес `0x4004be`, строка 233 находится по адресу `0x4004ce`, и.т.д.

```
[...]
232  result = positive_variable * arg1 + arg2;
0x4004be <+10>:  mov  0x200b64(%rip),%eax  # 0x601028 <positive_variable>
0x4004c4 <+16>:  imul -0x14(%rbp),%eax
0x4004c8 <+20>:  add  -0x18(%rbp),%eax
0x4004cb <+23>:  mov  %eax,-0x4(%rbp)

233  return result;
0x4004ce <+26>:  mov  -0x4(%rbp),%eax
[...]
```

Команда пошаговой отладки `step` скрывает запутанные переходы, происходящие уровнем ниже. В момент, когда пользователь хочет перейти к следующей строке программы, части, ответственной за работу в целевой системе, отправляется запрос на выполнение единственной инструкции программы и повторной остановки (это одна из операций, которую можно выполнить с помощью вызова `ptrace`). После получения информации об остановке выполнения программы, GDB запрашивает значение из регистра программного счетчика (program counter - PC) (другая операция, выполняемая частью, ответственной за работу в целевой системе), после чего происходит сравнение этого значения с диапазоном адресов, который ассоциирован с данной строкой в части, ответственной за обработку символов. Если значение программного счетчика находится за пределами этого диапазона, GDB оставляет программу в остановленном состоянии, устанавливает новый номер строки исходного кода и сообщает об этом пользователю. Если же значение программного счетчика все еще находится в диапазоне адресов текущей строки, GDB переходит к выполнению следующей инструкции с последующим повторением проверки, повторяя эту последовательность действий до тех пор, пока программный счетчик не станет указывать на другую строку. Преимущество этого простого алгоритма заключается в том, что он всегда работает правильно независимо от того, имеются ли в строке переходы, вызовы подпрограмм, и.т.д., при этом он не требует от GDB интерпретации всех всех деталей, относящихся к используемому набору машинных инструкций. Недостаток же заключается в большом количестве взаимодействий с целевой системой в ходе каждого отдельного шага, что может привести к заметному замедлению отладки при использовании некоторых встраиваемых целевых систем.

4.5. Переносимость

Ввиду необходимости постоянного доступа к физическим регистрам чипа в процессе работы, приложение GDB изначально проектировалось с учетом возможности переноса на широкий спектр систем. Однако, стратегия переносимости приложения с течением времени значительно изменилась.

Изначально программа GDB разрабатывалась аналогично всем остальным программам проекта GNU того времени; для разработки использовалось ограниченное подмножество функций языка C, причем при написании кода применялась комбинация макросов препроцессора и фрагментов файла Makefile для адаптации к специфической архитектуре процессора и операционной системе. Несмотря на то, что главной задачей проекта GNU было создание самодостаточной "Операционной системы GNU", запуск приложений должен был быть возможен в множестве существующих на тот момент систем; разработка ядра Linux начнется только спустя несколько лет. Сценарий оболочки с именем `configure` является первым ключевым звеном процесса. Он может выполнять множество различных действий, таких, как создание символьной ссылки для специфичного для используемой системы файла, позволяющей использовать стандартное имя заголовочного файла или формирование файлов на основе фрагментов, а наиболее важная работа, заключающаяся в сборке программы, осуществлялась с использованием файла Makefile.

Такие программы, как GCC и GDB предъявляют большие требования к процессу переноса на другие системы в сравнении с такими программами, как `cat` или `diff`, поэтому со временем действия по переносу GDB на другие платформы были разделены на три класса, причем для выполнения каждого действия использовался свой фрагмент файла Makefile и свой заголовочный файл.

- "Относящиеся к узлу объявления" используются для работы с системой, на которой выполняется GDB и могут включать такие объявления, как размеры целочисленных типов, применявшихся на узле. После изначальной реализации в виде создаваемых пользователями заголовочных файлов, стало ясно, что необходимые расчеты могут проводиться в ходе вызова сценарием `configure` небольших тестовых программ, которые компилируются с помощью того же компилятора, который будет использоваться для сборки приложения. Для этой цели может использоваться набор инструментов `autoconf` [aut12] и на сегодняшний день практически все инструменты проекта GNU, а также большинство (если не все) программы для Unix используют генерированные средствами `autoconf` сценарии конфигурации.
- "Относящиеся к целевой системе объявления" специфичны для машины, на которой будет выполняться отлаживаемая программа. В том случае, если целевая система аналогична системе узла, на котором выполняется приложение GDB, выполняется "непосредственная отладка", в противном случае выполняется "кросс-отладка", при которой используется ка-

кой-либо тип соединения двух систем. Относящиеся к целевой системе объявления могут быть разделены на два основных класса:

- "Объявления, относящиеся к архитектуре": Эти объявления описывают принцип дизассемблирования машинного кода, принцип обхода стека вызовов, а также инструкции-ловушки, которые следует выставлять в точках останова. Изначально реализованные с помощью макросов, они были преобразованы в обычный код на языке C, доступ к которому может быть получен с использованием объектов "gdbarch", о чем более подробно будет написано ниже.
- "Объявления, относящиеся непосредственно к системе": Эти объявления описывают специфику передачи аргументов при использовании вызовов `ptrace` (которые в значительной степени отличаются в различных Unix-системах), принцип поиска загруженных разделяемых библиотек, и другие подобные вещи, относящиеся исключительно к случаю непосредственной отладки. Эти объявления являются последним примером использования макросов в стиле 1980 годов, но большая часть этих объявлений уже генерируется средствами `autoconf`.

4.6. Структуры данных

Перед подробным рассмотрением составных частей приложения GDB, давайте рассмотрим основные структуры данных, с которыми работает GDB. Так как при разработке GDB используется язык программирования C, они реализуются в виде структур (`struct`) вместо объектов в стиле C++, но в большинстве случаев они рассматриваются как объекты, поэтому в данном случае мы также будем следовать практике разработчиков GDB и называть их объектами.

Точки останова

Точка останова является основным типом объекта, к которому пользователь может получить непосредственный доступ. Пользователь создает точку останова с помощью команды `break`, аргументы которой задают расположение (*location*), в качестве которого может использоваться имя функции, номер строки для исходного кода или адрес для машинного кода. GDB ставит в соответствие объекту точки останова небольшое целочисленное значение, которое пользователь может использовать впоследствии для работы с точкой останова. В рамках GDB точка останова представлена структурой языка C (`struct`) с множеством полей. Расположение точки останова преобразуется в машинный адрес, но при этом также сохраняется в оригинальном формате, ведь адрес может измениться, например, в том случае, когда программа повторно компилируется и загружается в рамках открытой сессии, поэтому требуется его повторный расчет.

Несколько типов объектов, а именно объекты точек наблюдения (`watchpoints`), точек перехвата (`catchpoints`) и точек трассировки (`tracepoints`), являющихся сходными по функциям с объектами точек останова, фактически используют структуру (`struct`) объектов точек останова. Это позволяет быть уверенным в том, что функции создания, изменения и удаления объектов постоянно будут находиться в работоспособном состоянии.

Термин "расположение" также относится к адресам в памяти, в которых будет установлена точка останова. В случаях использования `inline`-функций и шаблонов языка C++, можно столкнуться с ситуацией, когда единственная установленная пользователем точка останова может соответствовать нескольким адресам; например, каждая копия `inline`-функции требует отдельного расположения точки останова, которая была установлена в строке исходного кода тела этой функции.

Символы и таблицы символов

Таблицы символов относятся к ключевым структурам данных GDB и могут иметь достаточно большие размеры, иногда занимая по несколько гигабайт оперативной памяти. В некоторой степени такое поведение неизбежно; приложение больших размеров на языке C++ может содержать миллионы символов, при этом используя системные заголовочные файлы, которые в свою очередь также содержат миллионы дополнительных символов. Каждая локальная переменная, каждый именованный тип, каждый элемент перечисления - все это отдельные символы.

GDB использует множество обходных путей для сокращения объема таблиц символов, примерами которых являются такие обходные пути, как создание неполных таблиц символов (более подробно о них будет сказано позже), битовых полей в структурах (`struct`), и.т.д.

В дополнение к таблицам символов, которые по существу ставят символьные строки в соответствие адресам и данным о типах, GDB создает таблицы строк, с помощью которых поддерживается возможность перехода в двух направлениях; от номеров строк исходного кода к адресам в памяти и впоследствии от адресов памяти назад к строкам исходного кода. (Например, описанный ранее пошаговый алгоритм отладки не может работать без сопоставления адресов памяти и номеров строк исходного кода.)

Стековые фреймы

Процедурные языки программирования, для работы с которыми проектировался отладчик GDB, используют стандартную архитектуру времени исполнения, в рамках которой вызовы функций приводят к помещению программного счетчика в стек вместе с некоторой комбинацией из аргументов функций и локальных аргументов. Этот набор данных называется стековым фреймом (*stack frame*) или "фреймом" для краткости и в любой момент исполнения программы стек состоит из связанной последовательности фреймов. Подробности организации стекового фрейма значительно отличаются при переходе от использования одного чипа к другому, при этом они также зависят от используемых операционной системы, компилятора и параметров оптимизации.

Перенос GDB для использования при работе с новым чипом может потребовать разработки большого объема кода для анализа стека, так как программы (особенно при наличии ошибок, ведь именно в таких случаях пользователи наиболее часто запускают отладчик) могут быть остановлены в любом месте, причем фреймы могут быть неполными или частично перезаписанными самой программой. Что еще хуже, формирование стекового фрейма для каждого вызова функции замедляет приложение, поэтому проводящий хорошую оптимизацию компилятор при любом удобном случае упростит стековые фреймы или даже избавится от них, как это происходит при хвостовых вызовах.

Результат специфичного для чипа анализа стека, проводимого GDB, записывается путем формирования серий фреймовых объектов. Изначально GDB отслеживал фреймы, используя точное значение из регистра указателя фиксированных фреймов. Этот подход не позволял работать в случае применения вызовов `inline`-функций и других типов оптимизаций компилятора, поэтому в 2002 году разработчики GDB представили реализацию фреймовых объектов, которые записывают данные, полученные при исследовании каждого из фреймов, и связаны друг с другом, зеркалируя таким образом стековые фреймы приложения.

Выражения

Как и в случае с стековыми фреймами, GDB предполагает, что существует некая степень унификации всех поддерживаемых языков программирования и представляет все языки в виде древовидной структуры, созданной из узловых объектов. Набор типов узлов на самом деле является объединением всех типов выражений, свойственных всем различным языкам программирования; в отличие от компилятора, не существует причины, по которой пользователь не должен пытаться вычислить значение переменной языка Fortran из значения переменной языка C - при этом очевидно, что отличие значений этих переменных будет выражаться степенью числа два и в этот момент нам снова придется сказать "ага".

Значения

Результат вычисления сам по себе может быть более сложным, чем целочисленное значение или адрес в памяти, поэтому GDB также удерживает результаты вычисления в нумерованном списке

истории, на который позднее могут даваться ссылки в выражениях. Для поддержания работоспособности этой системы, GDB использует структуру данных значений. Структуры значений (*struct*) имеют множество полей, хранящих различные свойства; важные свойства представлены в форме указаний на то, является ли значение правым (*r-value*) или левым (*l-value*) (по отношению к левым значениям могут использоваться операции присваивания в языке C), а также указаний на то, может ли значение вычисляться при необходимости.

4.7. Часть, ответственная за работу с символами

Часть GDB, отвечающая за работу с символами главным образом предназначена для чтения исполняемого файла, извлечения из него любой найденной информации о символах и добавления этой информации в таблицу символов.

Процесс чтения начинается с библиотеки BFD. BFD является разновидностью универсальной библиотеки для обработки бинарных файлов и файлов с объектным кодом; работая в любой системе, она может читать и записывать файлы в оригинальном формате Unix *a.out*, формате COFF (используемом в System V Unix и MS Windows), формате ELF (используемом в современных системах Unix, GNU/Linux и большинстве встраиваемых систем), а также в некоторых других форматах. Внутренне устройство этой библиотеки представлено сложной структурой макросов языка C, которые преобразуются в код, в котором объединяются запутанные особенности форматов файлов объектного кода для множества различных систем. Представленная в 1990 году, библиотека BFD также используется ассемблером и линковщиком GNU, а ее способность формирования файлов объектного кода для любой целевой системы делает ее ключевым программным компонентом при кроссплатформенной разработке с использованием инструментов, предоставляемых проектом GNU. (Перенос библиотеки BFD на другие платформы является также первым ключевым шагом процесса переноса набора инструментов на новую целевую платформу.)

GDB использует библиотеку BFD исключительно для чтения файлов, а именно для извлечения блоков данных из исполняемого файла и помещения их в пространство памяти процесса GDB. После этого GDB может использовать собственные функции чтения данных, разделенные на два уровня. Первый уровень позволяет получить базовую информацию о символах или "минимум данных для символов", который представлен только самими именами символов, которые требуются линковщику для выполнения работы. Эти имена являются всего лишь строками с адресами и ничем больше; будем считать, что адреса в текстовых секциях соответствуют функциям, адреса в секциях данных соответствуют данным, и так далее.

Второй уровень позволяет получить подробную информацию о символах, которая обычно имеет свой собственный формат, отличающийся от базового формата исполняемого файла; например, информация в формате данных отладки DWARF хранится в секциях файла формата ELF со специальными названиями. В отличие от нее, информация в старом формате данных отладки *stabs*, применяемом в Berkley Unix, использует специально обозначенные символы из основной таблицы символов.

Код, предназначенный для чтения информации о символах, является в какой-то мере сложным для чтения, так как различные форматы информации о символах позволяют кодировать любые разновидности информации о типах, которая в свою очередь будет использоваться в исходной программе, но в рамках каждого из форматов это действие осуществляется своим уникальным способом. Часть GDB, ответственная за чтение данных символов, просто обрабатывает данные определенного формата и формирует символы GDB, руководствуясь предположением об их соответствии в рамках используемого формата.

Частичные таблицы символов

Для программы значительного размера (такой, как Emacs или Firefox) формирование таблицы символов может занять достаточно длительный промежуток времени, возможно даже несколько минут. Измерения четко указывают на то, что это время тратится не на чтение фала, как кто-либо мог предположить, а на формирование таблицы символов в пространстве памяти GDB. В этом случае приходится обрабатывать буквально миллионы небольших взаимосвязанных объектов, что занимает время.

Доступ к большей части информации о символах никогда не будет осуществлен в рамках сессии, так как она является локальной для функций, которые пользователь скорее всего никогда не будет использовать. Таким образом, в тот момент, когда отладчик GDB в первый раз извлекает информацию о символах программы, он производит поверхностный обзор информации о символах и ищет исключительно глобально видимые символы, записывая их в таблицу символов. Информация о символах для функции или метода в полном объеме извлекается только тогда, когда пользователь останавливает выполнение программы в этой функции.

Частичные таблицы символов позволяют GDB стартовать в течение нескольких секунд, даже при работе с программами значительного размера. (Символы разделяемых библиотек также загружаются динамически, но процесс их загрузки кардинально отличается. Обычно GDB использует специфичную для используемой системы технику получения оповещения о загрузке библиотеки, после чего формирует таблицу символов с функциями и соответствующими им адресами, полученными от динамического линковщика.)

Поддержка языков программирования

Поддержка языков программирования в основном заключается в реализации системы разбора выражений и вывода значений. Детали реализации системы разбора выражений зависят от языка программирования, но в общем случае она базируется на системе анализа грамматики Yacc, взаимодействующей со специально разработанным лексическим анализатором. В соответствии с целями разработки GDB, заключающимися в обеспечении гибкости при работе в интерактивном режиме, система разбора выражений не должна быть особенно строгой; например, если она сможет определить подходящий тип данных для выражения, она просто будет рассматривать случай использования этого типа без предъявления пользователю требования, заключающегося в необходимости добавления операции приведения или преобразования типа.

Так как система разбора выражений не должна обрабатывать объявления или декларации типов, она гораздо проще полноценной системы разбора выражений языка программирования. Аналогично, в случае вывода значений приходится иметь дело с множеством типов значений, которые следует выводить и обычно специфичная для языка программирования функция вывода данных может вызвать необходимый код для непосредственного выполнения задачи.

4.8. Часть, ответственная за работу в целевой системе

Часть, ответственная за работу в целевой системе, занимается управлением процессом исполнения программы и обрабатывает данные. В некотором смысле эта часть приложения GDB является полноценным низкоуровневым отладчиком; если вам хватает возможностей пошагового исполнения инструкций и создания необрабатываемых дампов памяти приложения, вы можете работать с GDB вообще без использования таблиц символов. (Так или иначе, вы можете оказаться в таком положении в том случае, если при выполнении функции библиотеки, символы которой были удалены из бинарного файла, исполнение программы будет остановлено.)

Векторы и стек целевой системы

Изначально часть GDB, ответственная за работу на целевой системе, была сформирована из набора специфичных для платформы файлов исходного кода, в рамках которых были реализованы ал-

горитмы вызова `ptrace`, запуска исполняемых файлов, и выполнения других подобных функций. Это решение не было достаточно гибким для использования при работе в рамках долговременных сессий отладки, когда пользователь может перейти от непосредственной к удаленной отладке, переключиться с отладки исполняемых файлов на анализ дампов памяти, после чего перейти к отладке работающих программ, подключать и отключать отладчик и выполнять другие аналогичные действия, поэтому в 1990 году John Gilmore провел повторное проектирование части GDB, ответственной за работу в целевой системе, направленное на выполнение всех специфичных для целевой системы операций посредством вектора целевой системы (*target vector*), являющегося по существу классом, на основе которого создаются объекты, каждый из которых описывает специфику типа целевой системы. Каждый вектор целевой системы реализован в форме структуры, состоящей из нескольких множеств указателей функций (обычно называемых "методами"), назначение которых варьируется от чтения и записи данных памяти и регистров до возобновления исполнения программы и установления параметров обработки разделяемых библиотек. В GDB существует около 40 векторов целевых систем, распределенных в диапазоне от часто используемого вектора целевой системы для Linux до малоизвестных векторов целевых систем, таких, как вектор для управления Xilinx MicroBlaze. Код поддержки дампов памяти использует вектор целевой системы, который получает данные путем чтения файла дампа памяти, при этом существует другой вектор целевой системы, читающий данные из исполняемого файла.

Обычно оказывается полезным смешивать методы нескольких векторов целевых систем. Представим случай вывода значения инициализированной глобальной переменной в Unix; перед началом исполнения программы вывод значения переменной будет работать, но в этот момент не будет процесса для чтения данных, поэтому байты данных должны быть получены из секции `.data` исполняемого файла. Таким образом, GDB использует векторы целевых систем для чтения исполняемых файлов и читает необходимые данные из бинарного файла. Но в процессе работы программы байты данных должны быть получены из адресного пространства процесса. Следовательно, в рамках GDB реализован "стек векторов целевых систем", в котором вектор целевой системы для работы с выполняющимися процессами помещается выше вектора для чтения данных из исполняемого файла при запуске процесса и перемещается вниз после завершения его работы.

В реальности стек целевых систем не обладает всеми ожидаемыми свойствами. Векторы целевых систем не являются действительно ортогональными друг другу; если вы работаете и с исполняемым файлом, и с выполняющимся процессом в рамках сессии, хотя и есть смысл отдавать предпочтение методам для работы с выполняющимся процессом, а не методам для работы с исполняемым файлом, практически никогда не имеет смысла делать наоборот. Поэтому в результате разработчики GDB ввели нотацию слоев (*stratum*), в рамках которой векторы целевых систем для работы с процессами объединяются в один слой, а векторы для работы с файлами объединяются в слой, находящийся ниже предыдущего, при этом векторы целевых систем могут добавляться наряду со вставкой и извлечением их из стека.

(Несмотря на то, что мэйнтайнеры GDB недолюбливают реализацию стека целевых систем, никто не предложил и не создал прототип более совершенной альтернативы.)

Gdbarch

Являясь программой, работающей напрямую с инструкциями центрального процессора, GDB требуется подробная информация об особенностях устройства чипа. Требуется информация о регистрах, размерах различных типов данных, размере и виде адресного пространства, используемом соглашении о вызовах функций, инструкции, которая будет вызывать исключение ловушки, и.т.д. Код на языке C для работы с этой информацией в GDB обычно занимает от 1,000 до 10,000 строк в зависимости от сложности архитектуры.

Изначально работа с этой информацией осуществлялась путем использования специфичных для целевой платформы макросов препроцессора, но по мере усложнения отладчика макросы все

больше увеличивались в размере и с течением времени длинные макроопределения были преобразованы в обычные функции языка C, вызываемые из макросов. Хотя это преобразование и было полезным, оно не могло применяться для различных вариантов архитектуры (ARM и ARM Thumb, 32-битная и 64-битная версии MIPS или x86, и.т.д.) и, что еще хуже, скоро должны были появиться многоархитектурные системы, в случае использования которых макросы не работали бы вообще. В 1995 году я предложил решение этой проблемы, заключающееся в использовании архитектуры на основе объектов и с 1998 года компания Cygnus Solutions начала финансирование работы Andrew Cagney, заключающейся в начальной реализации предложенных изменений. (Компания Cygnus Solutions была создана в 1989 году для коммерческой поддержки свободного программного обеспечения и в 2000 году была приобретена компанией RedHat.) Для завершения работы потребовалось несколько лет, причем в код также были внесены изменения от множества сторонних разработчиков и в итоге в ходе работы было изменено около 80,000 строк кода.

Новые реализованные конструкции стали называть объектами `gdbarch` и в данный момент эти объекты могут содержать до 130 методов и переменных, описывающих целевую архитектуру, но, несмотря на это, простая целевая архитектура может потребовать использования только части этих методов и переменных.

Для получения представления о том, чем отличаются старый и новый методы, рассмотрим объявление размера для типа данных `long double` архитектуры x86, равного 96 битам, из файла `gdb/config/i386/tm-i386.h` от 2002 года:

```
#define TARGET_LONG_DOUBLE_BIT 96
```

и из файла `gdb/i386-tdep.c` от 2012 года:

```
i386_gdbarch_init( [...] )
{
    [...]
    set_gdbarch_long_double_bit (gdbarch, 96);
    [...]
}
```

Управление исполнением

Сердцем GDB является цикл управления исполнением. Мы затрагивали его ранее при описании одиночных построчных переходов; алгоритм выполнял обход множества инструкций до момента нахождения инструкции, ассоциированной с другой строкой исходного кода. Этот цикл обхода называется `wait_for_inferior` или "wfi" для краткости.

Концептуально он находится внутри главного цикла обработки команд и используется только при вводе команд, которые направлены на продолжение исполнения программы. В момент, когда пользователь вводит команды `continue` или `step` и ожидает, не замечая никаких изменений, отладчик GDB на самом деле может быть загружен работой. В дополнение к описанному выше циклу пошагового перехода, программа может столкнуться с инструкциями-ловушками и сообщить об исключении GDB. Если исключение было сгенерировано из-за установки точки останова средствами GDB, производится проверка условия установки точки останова и в том случае, если условие не выполняется, ловушка убирается, осуществляется шаг по направлению к исходной инструкции, повторная установка ловушки, после чего выполнение программы возобновляется. Аналогично, в случае генерации сигнала GDB может принять решение о том, следует ли игнорировать его или произвести обработку одним из заранее заданных способов.

Все эти действия происходят под управлением цикла `wait_for_inferior`. Изначально это был простой цикл, ожидающий остановки выполнения программы на целевой системе и впоследствии принимающий решение о том, что сделать с полученными данными, но по мере переноса на различные системы потребовалась реализация специальных методов обработки данных и этот цикл увеличился в размере до тысячи строк, причем в нем по непонятным причинам начали применяться переходы `goto`. Например, с распространением вариантов систем Unix, не находилось понимающих тонкости их работы людей и не было возможности использования этих систем для тестирования и поиска регрессий, поэтому появилось обоснованное желание, заключающееся в осуществлении преобразования кода таким образом, чтобы не было изменено поведение приложения при работе с системами, на которые уже был осуществлен перенос, а переход через части цикла с помощью операторов `goto` был в данном случае самой простой тактикой.

Единственный большой цикл также был проблемой для асинхронной обработки или отладки многопоточных программ, при работе с которыми пользователь может изъять желание запускать или останавливать отдельные потоки, причем в этом случае остальные потоки программы должны были выполняться.

Преобразование к ориентированной на события модели заняло несколько лет. Я разделил цикл `wait_for_interior` на части в 1999 году, представив структуру управления состоянием выполнения приложения, предназначенную для замены набора локальных и глобальных переменных и преобразования беспорядочных переходов в небольшие независимые друг от друга функции. В то же время Elena Zanonni и другие разработчики представили очереди событий, которые позволяли получать и пользовательский ввод, и уведомления от цикла.

Протокол удаленной отладки

Несмотря на то, что векторы целевых архитектур GDB позволяют реализовать множество различных методов для управления процессом исполнения программы на удаленном компьютере, для этой цели у нас есть отдельный предпочтительный протокол. Для него не придумано определенного названия, поэтому для обозначения обычно используются термины: "удаленный протокол" ("remote protocol"), "удаленный протокол GDB" ("GDB remote protocol"), "удаленный последовательный протокол" (также используется аббревиатура "RSP", расшифровывающаяся как "remote serial protocol"), "удаленный протокол .c" ("remote.c protocol", в соответствии с расширением файла исходного кода с его реализацией) или иногда "протокол-заглушка" ("stub protocol") для указания на то, что реализация протокола осуществлена на целевой системе.

Основной протокол достаточно прост и отражает желание его реализации для работы с небольшими встраиваемыми системами 1980-х годов, объемы памяти которых измерялись в килобайтах. Например, в рамках протокола пакет `$g` запрашивает данные всех регистров и ожидает ответа, содержащего все байты данных из этих регистров, в итоге передаются все данные - предполагается, что количество регистров, их размер и порядок следования данных совпадают с используемыми в рамках проекта GDB соглашениями.

Протокол ожидает одиночного ответа на каждый отправленный пакет и предполагает, что соединение является надежным, добавляя контрольные суммы только к отправляемым пакетам (таким образом, пакет `$g` при отправке по сети на самом деле будет выглядеть как `$g#67`).

Несмотря на то, что существует только ограниченное количество обязательных типов пакетов (соответствующих половине методов вектора целевой системы, которые наиболее важны), по прошествии многих лет было добавлено большое количество дополнительных, необязательных пакетов для поддержки всех функций, начиная с функций для работы с аппаратными точками останова и заканчивая функциями для работы с точками трассировки и разделяемыми библиотеками.

На самой же целевой системе реализация удаленного протокола может быть представлена в широком диапазоне форм. Протокол в полной мере документирован в руководстве GDB и это значит, что имеется возможность создания реализации, которая не будет обременена условиями лицензии GNU и на самом деле многие производители оборудования уже реализовали код, который позволяет работать с удаленным протоколом GDB как во время лабораторных испытаний, так и во время эксплуатации оборудования. Система IOS компании Cisco, под управлением которой работает большая часть выпускаемого этой компанией сетевого оборудования, является широко известным примером.

Реализация протокола на целевой системе обычно называется "отладочная заглушка" или просто "заглушка" для того, чтобы подчеркнуть тот факт, что она выполняет не так много работы самостоятельно. Исходные коды GDB содержат несколько примеров реализации подобных заглушек, которые обычно реализуются в форме примерно 1,000 строк низкоуровневого исходного кода на языке C. На совершенно новой системе без установленной ОС код заглушки должен установить собственные обработчики для перехвата аппаратных исключений и, что более важно, для перехвата инструкций-ловушек. Также потребуется драйвер последовательного порта, если сеть организуется посредством последовательного соединения. Сама работа протокола достаточно проста, так как все необходимые пакеты содержат одиночные символы, которые могут быть декодированы при помощи оператора switch.

Другим подходом к реализации удаленного протокола является создание "спрайта" для обмена данными между копией GDB и удаленным отлаживаемым аппаратным обеспечением, представленным устройствами JTAG, устройствами "wiggler" для интерфейса JTAG, и.т.д. Обычно эти устройства имеют библиотеку, которая может использоваться на компьютере, физически соединенным с целевым устройством, при этом обычно API библиотеки не совместим в внутренней архитектурой GDB. Таким образом, хотя и существуют примеры непосредственного вызова функций библиотек из GDB, наиболее простым зарекомендовавшим себя способом является запуск спрайта в качестве независимой программы, которая понимает удаленный протокол и преобразует пакеты в вызовы функций пред назначенной для работы с устройством библиотеки.

GDBServer

Исходные коды GDB включают в свой состав одну завершенную и работоспособную реализацию удаленного протокола для использования на целевой системе: GDBserver. GDBserver является скомпилированной программой, выполняющейся в целевой операционной системе и осуществляющей контроль над другими выполняющимися в целевой операционной системе программами, которая использует возможности непосредственной отладки в ответ на пакеты, полученные с применением удаленного протокола. Другими словами, эта программа выступает в роли специфического прокси-сервера для непосредственной отладки.

GDBserver не выполняет каких-либо действий, которые не может выполнять сам отладчик GDB; если в вашей целевой системе может выполняться GDBserver, то теоретически в ней сможет работать и GDB. Однако, приложение GDBserver в 10 раз меньше и ему не требуется осуществлять управление таблицами символов, поэтому оно очень хорошо подходит для работы со встраиваемыми системами на основе GNU/Linux, а также с другими подобными системами.

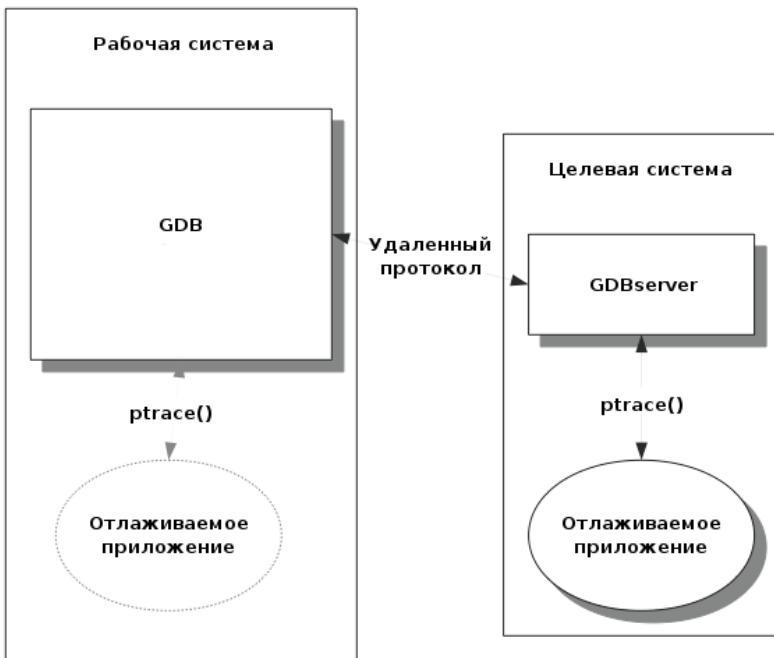


Рисунок 4.2: GDBserver

Приложения GDB и GDBserver частично используют общий код, но хотя идея инкапсуляции специфичных для ОС функций контроля над процессами и является очевидной, существуют сложности, связанные с разделением подразумеваемых зависимостей в рамках GDB и процесс разделения идет медленно.

4.9. Интерфейсы GDB

По своей сути GDB является отладчиком с интерфейсом командной строки. В течение долгого времени люди пробовали различные схемы для реализации на его основе графического многооконного интерфейса отладчика, но, несмотря на все потраченное время и приложенные усилия, ни один из этих интерфейсов так и не стал общепринятым.

Интерфейс командной строки

Интерфейс командной строки использует стандартную библиотеку `readline` проекта GNU для посимвольной обработки пользовательского ввода. Библиотека `readline` заботится о таких вещах, как редактирование строк и завершение команд; пользователь может выполнять такие действия, как использование клавиш перемещения курсора для перехода к предыдущей строке и исправления символа.

После этого GDB принимает возвращаемую библиотекой `readline` команду и производит ее поиск в каскадной структуре таблиц команд, в которой при каждом успешном обнаружении управляющего слова производится выбор дополнительной таблицы. Например, команда `set print elements 80` затрагивает три таблицы; первой является таблица всех доступных команд, второй - таблица параметров команды `set`, а третьей - таблица параметров вывода значений, в соответствии с которой `elements` является одним из ограничений, налагаемых на число объектов, выводимых при рассмотрении таких конструкций, как строка или массив. После того, как с помощью каскадной структуры была вызвана соответствующая функция для обработки команды, она получает контроль над процессом выполнения команды и обработка аргументов является исключительно ее задачей. Некоторые команды, такие, как `run` обрабатывают свои аргументы по аналогии со стандартом `argc/argv`, традиционным для языка C, в то время, как другие команды, такие, как `print` предполагают, что оставшаяся часть строки является единым выражением языка програм-

мирования и передают всю строку специфичной для языка программирования системе разбора синтаксических конструкций.

Машинный интерфейс

Одним способом реализации графического интерфейса для отладки является использование GDB в роли "обработчика команд" реализующей графический интерфейс программы, которая занимается преобразованием событий нажатия кнопок мыши в команды и форматированием выводимых результатов для их показа в окнах. Описанный подход использовался несколько раз в рамках таких приложений, как KDbg и DDD (Data Dispaly Debugger), но он не был идеальным по той причине, что иногда результаты форматировались для удобства чтения человеком, причем в этих случаях отбрасывались подробности выполнения команды и делалась ставка на возможность человека мыслить в определенном контексте.

Для решения этой проблемы в GDB реализован альтернативный "пользовательский" интерфейс, известный как машинный интерфейс (Machine Interface или MI для краткости). По своей сути это еще один интерфейс командной строки, но и команды и результаты их выполнения в данном случае используют дополнительные синтаксические конструкции для явного указания их особенностей - каждый аргумент помещен в кавычки и сложные выводимые структуры содержат разделители для подгрупп и имен параметров частей их компонентов. В дополнение команды машинного интерфейса могут использовать префиксы с идентификаторами последовательности, которые будут повторяться при выводе результатов выполнения команд для гарантии того, что результаты выполнения команд будут поставлены в соответствие правильным командам.

Для того, чтобы ознакомится с различиями двух форм вывода данных ниже приведен пример использования обычной команды step и ответ GDB:

```
(gdb) step
buggy_function (arg1=45, arg2=92) at ex.c:232
232 result = positive_variable * arg1 + arg2;
```

При использовании машинного интерфейса ввод и вывод содержит больше подробностей, причем данные оформлены таким образом, чтобы сторонне программное обеспечение имело возможность более точно произвести их разбор:

```
4321-exec-step
4321^done,reason="end-stepping-range",
  frame={addr="0x0000000004004be",
    func="buggy_function",
    args=[{name="arg1",value="45"}, {name="arg2",value="92"}],
    file="ex.c",
    fullname="/home/sshebs/ex.c",
    line="232"}
```

Окружение разработки Eclipse [[ecl12](#)] является наиболее известным примером клиента машинного интерфейса.

Другие пользовательские интерфейсы

Дополнительные пользовательские интерфейсы представлены версией на основе tcl/tk с названием GDBtk или Insight, а также версией на основе curses с названием TUI, изначально реализованной компанией Hewlett-Packard. GDBtk является обычным многопанельным графическим интерфей-

сом, созданным с использованием библиотеки tk, в то время, как TUI является интерфейсом, созданным путем разделения областей текстового вывода на экране.

4.10. Процесс разработки

Мэйнтейнеры

Как и в случае любой оригинальной программы проекта GNU, процесс разработки GDB следует "соборной" модели. Изначально разработанное Richard Stallman, приложение GDB прошло ряд "мэйтейнеров", каждый из которых был по совместительству архитектором, исследователем патчей и ответственным за выпуск релизов лицом, имеющим доступ к репозиторию исходного кода, который предоставлялся только ограниченному числу работников компании Cygnus.

В 1999 году проект GDB мигрировал в публичный репозиторий исходного кода, при этом команда мэйтейнеров увеличилась до нескольких десятков человек, которым помогали сторонние разработчики, наделенные привилегиями внесения изменений в исходный код. Это изменение процесса разработки позволило значительно ускорить его, увеличив количество еженедельных изменений в исходном коде с 10 до 100 и более.

Тестирование, тестирование

Так как GDB является системо-зависимым приложением, перенесенным на множество систем в диапазоне от очень малых до очень больших и поддерживающим сотни команд, параметров и стилей использования, даже для опытного разработчика GDB достаточно сложно предвидеть все эффекты, вызванные изменениями в коде.

Для этой цели и предназначен набор тестов. Набор тестов состоит из множества тестовых программ, комбинируемых со сценариями инструмента тестирования `expect`, использующими фреймворк тестирования на основе tcl с названием DejaGNU. Базовая модель тестирования основана на том, что каждый сценарий использует отладчик GDB таким образом, как это делается при отладке программы, отправляя команды и после их выполнения сравнивая вывод с шаблонами с использованием регулярных выражений.

Набор тестов также поддерживает возможность запуска кросс-отладки как для работающего аппаратного обеспечения, так и для симуляторов, а также позволяет использовать тесты, специфичные для отдельной архитектуры или конфигурации.

На конец 2011 года набор тестов состоит из 18,000 тестов, представленных тестами базовых функций, тестами, специфичными для языков программирования, тестами, специфичными для архитектур, а также тестами машинного интерфейса. Большая часть этих тестов представлена обобщенными тестами, которые могут выполняться при любой конфигурации. Лицам, вносящим изменения в код GDB, следует выполнить тесты из набора по отношению к измененному исходному коду и убедиться в отсутствии регрессий, причем для каждой новой возможности должны реализоваться новые тесты. Однако, из-за того, что ни у кого нет доступа ко всем платформам, которые могут быть затронуты при внесении изменения в исходный код, очень редко удается достичь полного отсутствия неудачно завершенных тестов; наличие 10-20 неудачно завершенных тестов обычно вполне допустимо для создания копии исходного кода, предназначенный для непосредственной отладки, при этом для некоторых встраиваемых систем количество неудачно завершенных тестов может быть большим.

4.11. Выученные уроки

Открытый процесс разработки является выигрышным решением

Приложение GDB изначально являлось экземпляром программного обеспечения, созданного в процессе применения "соборной" модели разработки, в рамках которой мэнтейнер осуществляет непосредственное управление исходным кодом, а весь остальной мир наблюдает за прогрессом разработки только при периодической публикации копий исходного кода. Применение такой модели обосновывалось относительным непостоянством в отправке патчей, но на самом деле закрытый процесс разработки сам по себе препятствовал созданию патчей. После перехода к открытому процессу разработки количество патчей стало таким большим, каким не было никогда ранее, причем качество кода осталось таким же высоким, как и было ранее, а в некоторых случаях повысилось.

Создавайте план, но ожидайте его изменения

Процесс разработки приложения с открытым исходным кодом по своей сути является хаотичным, так как отдельные лица в течение определенного промежутка времени работают над кодом, после чего прекращают работу, возлагая обязанности по продолжению разработки на других.

Однако, в данном случае все еще уместно создавать план разработки и публиковать его. Он может помочь в координации действий разработчиков в том случае, если они работают над реализацией взаимосвязанных возможностей, он может быть показан потенциальным спонсорам, а также он может помочь желающим поучаствовать в развитии проекта лицам в размышлениях о том, чем именно им стоит заняться.

При составлении плана не следует обозначать точные даты и промежутки времени; даже в том случае, если кто-либо проявляет энтузиазм при работе в определенном направлении, очень мало-вероятно то, что люди смогут гарантировать работу в течение полного рабочего дня в течение такого длительного промежутка времени, который позволит завершить работу до выбранной даты.

По этой причине не нужно цепляться за план в случае истечения указанных в нем сроков. В долговременной перспективе в рамках разработки проекта GDB был составлен план реструктуризации приложения и выделения библиотеки `libgdb` с четко заданным API, которая могла бы связываться со сторонними приложениями (в особенности с приложениями, реализующими графические интерфейсы); при этом даже был изменен процесс сборки с целью компиляции библиотеки `libgdb.a` на промежуточном шаге. Несмотря на то, что идея о создании библиотеки периодически озвучивалась впоследствии, преимущества интегрированной среды разработки Eclipse и машинного интерфейса отодвинули обоснование необходимости создания библиотеки на второй план, а в январе 2012 года мы отвергли концепцию библиотеки и на данный момент занимаемся удалением теперь уже не нужных фрагментов кода.

Все было бы идеально, если бы мы обладали незаурядным умом

После обзора некоторых произведенных нами изменений, вы можете подумать: "Почему с самого начала все не было сделано правильно?" Ну, мы были просто не достаточно умны.

Конечно же, мы могли ожидать, что отладчик GDB станет чрезвычайно популярным и будет перенесен на десятки и десятки архитектур для осуществления и непосредственной и удаленной отладки. Если бы мы знали это, мы начали бы разработку с создания объектов `gdbarch` вместо усовершенствования старых макроопределений и глобальных переменных в течение нескольких лет; то же самое можно сказать и о векторах целевых систем.

Конечно же, мы могли догадываться о том, что GDB будет использоваться с графическими интерфейсами. Кроме того, в 1986 году как Mac, так и X Window System уже существовали в течение двух лет! Вместо проектирования традиционного интерфейса командной строки, мы могли создать интерфейс для асинхронной обработки событий.

При этом в реальности урок заключался не в том, что разработчики GDB были глупы, а в том, что мы, возможно, не были достаточно прозорливы и не имели представления о том, как должен развиваться проект GDB. В 1986 году вообще не было очевидным тот факт, что использующий мышь оконный интерфейс начнет применяться повсеместно; в том случае, если бы первая версия GDB была в совершенстве адаптирована для использования с графическим интерфейсом, мы бы выглядели как гении, но это была бы всего лишь случайная удача. Вместо этого, улучшая работу GDB в одной ограниченной области, мы сформировали пользовательскую базу, которая позволила нам в будущем увеличить темпы разработки и повторного проектирования.

Следует учиться жить с незавершенными переходами к использованию новых технологий

Пытайтесь довести до конца переход к использованию новых технологий, хотя этот процесс и может занять некоторое время; будьте готовы работать в условиях незавершенных переходов.

На саммите разработчиков GCC в 2003 году Zack Weinberg сожалел о "незавершенных переходах к использованию новых технологий" в рамках проекта GCC, при осуществлении которых новая инфраструктура была введена в строй в то время, как старая инфраструктура не могла быть свернута. В проекте GDB тоже присутствуют такие переходы, но в то же время мы можем привести в пример несколько переходов, которые были успешно завершены, таких, как переход к использованию векторов целевых систем и переход к использованию `gdbarch`. Несмотря на это, для их реализации потребовалось несколько лет и в течение этого времени приходилось поддерживать отладчик в работоспособном состоянии.

Страйтесь не сильно привязываться к коду

В ситуации, когда вы в течение длительного промежутка времени работаете с отдельным фрагментом кода, причем вы работаете с важной программой, достаточно легко привязаться к этому коду, причем в этом случае вы будете даже формировать свои мысли в соответствии с этим кодом, а не наоборот.

Страйтесь избегать таких ситуаций.

Любой фрагмент кода создавался на основе осознанных решений: некоторые из решений принятые в большей степени под впечатлением от чего-либо, некоторые - в меньшей степени. Умное принятие в 1991 году решения, заключающееся в уменьшении объема используемой оперативной памяти, является лишним усложнением кода в 2011 году, когда вполне доступны объемы оперативной памяти, исчисляющиеся гигабайтами.

Когда-то отладчик GDB мог работать с суперкомпьютером Gould. Когда примерно в 2000 году была выведена из строя последняя подобная машина, пропал весь смысл поддержки этой системы. Данный эпизод является иллюстрацией процесса удаления кода из состава GDB, причем на сегодняшний день при выпуске большинства релизов та или иная часть кода считается устаревшей и удаляется.

Фактически существует список радикальных изменений, которые уже запланированы или реализуются и варьируются от применения языка Python для реализации сценариев до поддержки отладки приложений на многопроцессорных системах с высокой степенью параллелизма и перехода к использованию языка программирования C++. Для реализации этих изменений могут потребоваться годы; поэтому у нас есть еще больше оснований для начала работы над их реализацией прямо сейчас.

5. Компилятор Glasgow Haskell

Глава 5 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Разработка компилятора Glasgow Haskell Compiler (GHC) стартовала в начале 1990-х годов как часть исследовательского проекта, финансируемого правительством Великобритании, с намерением достичь нескольких целей:

- Создать свободно доступный, надежный и переносимый компилятор для языка Haskell, который должен генерировать высокопроизводительный код;
- Обеспечить модульную основу, которую могли бы расширять и развивать другие исследователи;
- Узнать, как ведут себя реальные программы, которые мы можем разрабатывать и создавать с помощью лучших компиляторов.

Компилятору GHC в настоящее время более 20 лет, и с момента своего создания он постоянно и активно разрабатывается. Сегодня версии компилятора GHC загружают сотни тысяч людей, онлайновый репозиторий библиотек Haskell содержит более 3000 пакетов, компилятор GHC используется для обучения языку Haskell на многих курсах высшей школы, и все больше появляется вариантов языка Haskell, используемых в коммерции.

За время своего существования у компилятора GHC в целом было два - три активных разработчика, хотя количество тех, кто добавил свой код в разработку компилятора GHC, исчисляется сотнями. Хотя конечной целью для нас, главных разработчиков компилятора GHC, является проведение исследований, а не создание кода, мы считаем, что разработка компилятора GHC является важным предварительным условием: результаты исследований снова возвращаются в компилятор GHC, так что компилятор GHC затем может использоваться в качестве основы для дальнейшего исследования, которое строится на предыдущих идеях. Более того, важно, чтобы компилятор GHC был изделием промышленного уровня, поскольку тем самым обеспечивается большая правдоподобность результатов проводимых с ним исследований. Таким образом, хотя компилятор GHC переполнен передовыми научными идеями, много усилий затрачивается на то, чтобы его можно было использовать в производственных целях. Между этими двумя, казалось бы, противоречивыми целями, часто возникает некоторая напряженность, но, по большому счету, мы нашли направление, которое является приемлемым как с точки зрения исследований, так и с точки зрения производственных задач.

В этой главе мы хотим дать обзор архитектуры компилятора GHC и сосредоточиться на нескольких ключевых идеях, которые оказались успешными в компиляторе GHC (и на нескольких тех, которые успеха не имели). Надеюсь, что из следующих нескольких страниц вы получите некоторое представление о том, как нам на протяжении более 20 лет удалось сохранить активным большой программный проект, причем он не разрушится под своим собственным весом, что, как правило, происходит в случае, когда команда разработчиков очень маленькая.

5.1. Что такое язык Haskell?

Язык Haskell является функциональным языком программирования, который определен согласно документу, известному как «Haskell Report» («Отчет по языку Haskell»), последней версией которого является документ «Haskell 2010» [Mar10]. Язык Haskell был создан в 1990 году несколькими представителями академического научно-исследовательского сообщества, заинтересованного в функциональных языках, с целью решить проблему отсутствия общего языка, который можно было бы использовать как субъект для своих исследований.

Две особенности языка Haskell выделяют его из множества других языков программирования:

- Это *чисто функциональный* (*purely functional*) язык. То есть, функции не могут иметь побочные эффекты или изменять данные; для заданного набора входных данных (аргументов)

функция всегда выдает один и тот же результат. Что касается кода (и, на наш взгляд, создания кода), преимущества этой модели очевидны, но добавление ввода/вывода в чисто функциональную среду потребовало серьезного исследования. К счастью было обнаружено элегантное решение в виде *монад* (*monads*), что позволило не только аккуратно интегрировать ввод/вывод в чисто функциональный код, но ввести новые мощные абстракции, которая произвели революцию в кодировании на языке Haskell (что впоследствии также оказало влияние на другие языки).

- Это язык с *отложенными вычислениями* (*lazy*). Это относится к стратегии вычислений в языке: в большинстве языков используются *строгие вычисления* (*strict*), когда аргументы функции вычисляются до вызова функции, тогда как в языке Haskell аргументы функции передаются *невычисленными* (*unevaluated*) и вычисляются только тогда, когда они потребуются. Что касается программ, то эта особенность языка Haskell также имеет свои преимущества, но прежде всего она служит барьером для предотвращения появления в языке нефункциональных свойств: такие свойства принципиально не могут сочетаться с семантикой отложенных вычислений.

Язык Haskell также является *строго типизированным* (*strongly-typed*), хотя он и поддерживает создание *производных типов* (*type inference*), что означает, что аннотации типов редко бывают необходимы.

Те, кому интересна подробная история языка Haskell, могут прочитать ее в [HHPW07].

5.2. Общий взгляд на проект

На самом высоком уровне, компилятор GHC можно разделить на следующие три отдельные части

- Сам компилятор. По сути, это программа на языке Haskell, работа которой заключается в преобразовании исходного кода на языке Haskell в исполняемый машинный код.
- Загружаемые библиотеки. GHC поставляется с набором библиотек, которые мы называем загрузочными библиотеками, поскольку они представляют собой библиотеки, от которых зависит сам компилятор. Наличие этих библиотек в дереве исходного кода означает, что компилятор GHC может сам себя развертывать (bootstrap). Некоторые из этих библиотек очень тесно связаны с компилятором GHC, поскольку в них реализованы низкоуровневые функции, такие как тип `Int`, в терминах примитивов, которые определяются компилятором и системой времени выполнения. Другие библиотеки являются библиотеками более высокого уровня и они менее зависимы от компилятора, например, библиотека `Data.Map`.
- Система времени выполнения RTS (Runtime System). Это большая библиотека кода на языке C, который обрабатывает все задачи, связанные с *выполнением откомпилированного кода языка Haskell*, в том числе со сборкой мусора, планированием потоков выполнения, профилированием, обработкой исключений и так далее. Система RTS компонуется с каждой откомпилированной программой на языке Haskell. На систему RTS была потрачена значительная часть усилий по разработке, которые были вложены в компилятор GHC, а проектные решения, реализованные в ней, относятся к наиболее сильным сторонам языка Haskell, например, его эффективной поддержке одновременного выполнения операций и распараллеливания. Мы опишем систему RTS более подробно в разделе 5.5.

По сути, эти три отдельные части в точности соответствуют трем подкаталогам дерева исходных кодов компилятора GHC: `compiler`, `libraries` и `rts`, соответственно

Мы здесь не будем тратить много времени на обсуждение загрузочных библиотек, поскольку они с архитектурной точки зрения малоинтересны. Все ключевые проектные решения воплощены в компиляторе и системе времени выполнения, поэтому мы посвятим оставшуюся часть этой главы обсуждению этих двух компонентов.

Оцениваем размер кода

В последний раз мы подсчитывали количество строк в компиляторе GHC в 1992 году — см. «The Glasgow Haskell compiler: a technical overview» («Компилятор Glasgow Haskell: технический обзор»), обзор на конференции JFIT, 1992, поэтому интересно посмотреть, как с тех пор все изменилось. На рис.5.1 приведены данные о количестве строк кода в компиляторе GHC в его основных компонентах; текущие значения сравниваются с теми, что были в 1992 году.

Модуль	Строки (1992 г.)	Строки (2011 г.)	Увеличение
Синтаксический анализ	1055	4098	3,9
Модуль переименований	2828	4630	1,6
Проверка типов	3352	24097	7,2
Сокращение синтаксического разнообразия языка	1381	7091	5,1
Основные преобразования	1631	9480	5,8
Преобразования STG	814	840	1
Распараллеливания данных в языке Haskell	—	3718	—
Генерация кода	2913	11003	3,8
Генерация нативного кода	—	14138	—
Генерация кода LLVM	—	2266	—
Интерактивная среда GHCi	—	7474	—
Абстрактный синтаксис языка Haskell	2546	3700	1,5
Ядро языка	1075	4798	4,5
Язык STG	517	693	1,3
Язык C-- (был язык Abstract C)	1416	7591	5,4
Представление идентификаторов	1831	3120	1,7
Представление типов	1628	3808	2,3
Прелюдия определений	3111	269	0,9
Утилиты	1989	7878	3,96
Профилирование	191	367	1,92
Всего в компиляторе	28275	139955	4,9
<i>Система времени выполнения</i>			
Весь код на языках С и C--	43865	48450	1,10

Рис.5.1: Количество строк в компиляторе GHC, прошлое и настоящее

В этих цифрах есть несколько примечательных особенностей:

- Несмотря на почти 20 летнюю безостановочную разработку компилятор увеличился в размерах только в 5 раз, от приблизительно 28000 строк и до 140000 строк кода на языке Haskell. При добавлении нового кода мы со всей страстью обновляем старый код, поддерживая основной код настолько обновленным, насколько это возможно.
- Есть несколько новых компонентов, хотя их размер равен приблизительно 28000 новых строк. Большая часть новых компонентов связана с генерацией кода: генераторы нативного кода для различных процессоров и генератор кода LLVM. (Бывший «Low Level Virtual Machine», проект LLVM, в состав которого входил универсальный генератор кода, позволяющий создавать код для различных процессоров. Дополнительную информацию смотрите в <http://llvm.org/>, и в главе о LLVM в первом томе сборника «Архитектура приложений с

открытым исходным кодом»). Инфраструктура интерактивного интерпретатора GHCi также добавила более 7000 строк кода.

- Наибольшее увеличение размера произошло в единственном компоненте — модуле проверки типов, где было добавлено более 20 000 строк кода. Это неудивительно, учитывая, что большая часть недавних исследований с использованием GHC была связана с расширением системы типов (например, типы GADT [PVWW06] и семейства типов Type Families [CKP05]).
- Много кода было добавлено в основном компоненте Main, и это отчасти связано с тем, что ранее был скрипт на Perl, называемый «драйвером» и имеющим 3000 строк кода, который был переписан на языке Haskell и был перемещен в другое место в компиляторе GHC, а также поскольку была добавлена возможность компиляции нескольких модулей.
- Система времени выполнения практически не увеличилась: несмотря на то, что появилось много новых функциональных возможностей и было выполнено портирование на другие платформы, увеличение размера кода составляет не более, чем 10%. Приблизительно в 1997 году мы ее полностью переписали.
- В компиляторе GHC есть сложная система сборки, которая сегодня включает в себя около 6000 строк кода для GNU Make. Именно она была полностью переписана четвертый раз около двух лет назад, причем при каждом переписывании размер кода сокращался.

Компилятор

Мы можем выделить в компиляторе следующие три части:

- *Менеджер компиляции (compilation manager)*, который отвечает за компиляцию нескольких исходных файлов на языке Haskell. Задача менеджера компиляции состоит в том, чтобы выяснить, в каком порядке компилировать отдельные файлы, и решить, какие модули не нужно перекомпилировать, потому что ни одна из их зависимостей не была изменена с того момента, когда они были последний раз скомпилированы.
- Собственно *компилятор языка Haskell (Haskell compiler)*, который осуществляет компиляцию отдельного исходного файла на языке Haskell. Внутри компилятора GHC мы обозначаем его как Hsc. Как вы догадываетесь, здесь происходит большая часть действий. Результат работы Hsc зависит от того, какой выбран генератор: ассемблер, код на языке LLVM или байт-кода.
- *Конвейер (pipeline)*, который отвечает за объединение вместе с Hsc всех внешних программ, необходимых для компиляции файла с исходным кодом на языке Haskell в объектный код. Например, файлу с исходным кодом на языке Haskell может потребоваться предварительная обработка с препроцессором C перед тем, как он будет передан Hsc, а результат работы Hsc, который, как правило, представляет собой файл на языке ассемблера, для создания объектной файла должен быть передан в ассемблер.

Компилятор является не просто исполняемым файлом, который выполняет эти функции; он сам является библиотекой с большим интерфейсом API, который можно использовать для создания других инструментальных средств, работающих с исходным кодом на языке Haskell, например, среды разработки IDE и аналитических инструментальных средств.

Компиляция кода на языке Haskell

Как и в большинстве компиляторов, компиляция файла с исходным кодом на языке Haskell выполняется как последовательность фаз, причем результат работы каждой фазы передается на вход следующей фазы. Общая структура различных фаз показана на рис.5.2.

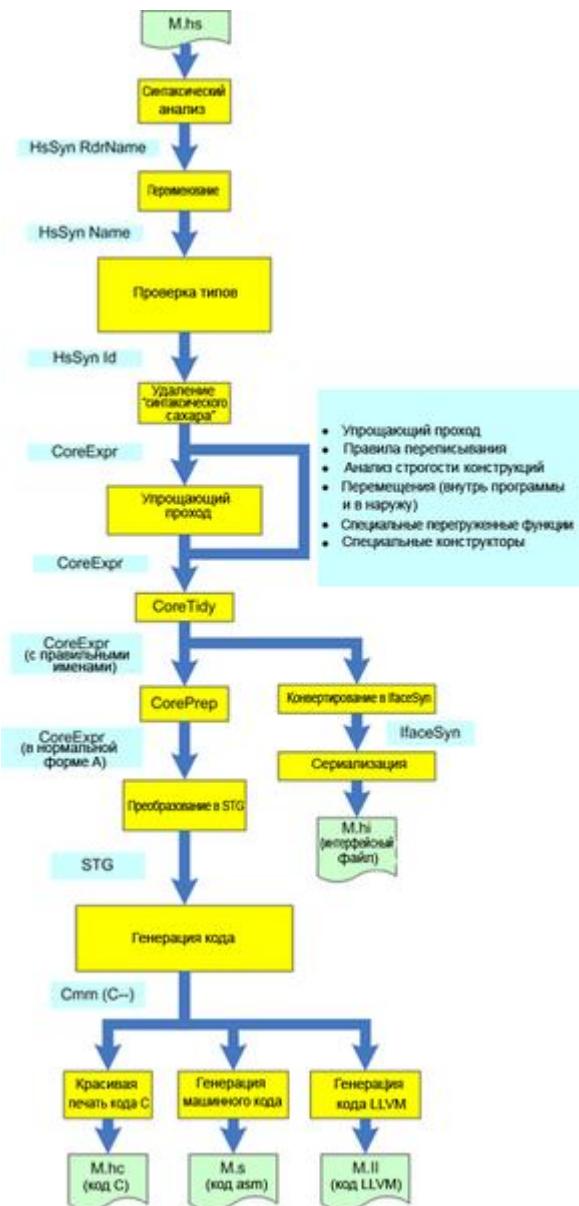


Рис.5.2: Фазы работы компилятора (картинка кликабельна)

Синтаксический разбор

Мы начинаем в традиционном стиле с синтаксического разбора, при котором в качестве входного берется файл с исходным кодом на языке Haskell, а на выходе создается абстрактный синтаксис. В компиляторе GHC тип данных абстрактного синтаксиса `HsSyn` параметризуется при помощи типов идентификаторов, которые в нем есть, с тем, чтобы для некоторого типа идентификаторов τ дерево абстрактного синтаксиса имело тип `HsSyn τ`. Это позволяет добавлять дополнительную информацию к идентификаторам по мере того, как программа проходит через различные фазы компилятора, и одновременно использовать один и тот же тип деревьев абстрактного синтаксиса.

Результатом выполнения синтаксического разбора является абстрактное синтаксическое дерево, в котором идентификаторы являются простыми строками, которые мы называем `RdrName`. Таким образом, абстрактный синтаксис, создаваемый парсером, имеет тип `HsSyn RdrName`.

В компиляторе GHC используются инструментальные модули `Alex` и `Happy` для генерации кода при лексическом и синтаксическом анализе, соответственно, которые являются аналогами инструментальных средств `lex` и `yacc` для языка C.

Парсер компилятора GHC является чисто функциональным. В действительности, интерфейс API из библиотеки GHC предоставляет чистую функцию, называемую `parser`, которая берет строку `String` (и некоторые другие данные) и возвращает либо разобранный абстрактный синтаксис, либо сообщение об ошибке.

Переименование

Переименование является процессом разрешения (присвоения новых имен — прим.пер.) всех идентификаторов, имеющихся в исходном коде языка Haskell, в полностью квалифицированные имена, одновременного определения того, выходят ли идентификаторы за область своей видимости, и, если это имеет место, создания соответствующих пометок об ошибках.

В языке Haskell можно в модуле повторно экспортировать идентификатор, который импортируется из другого модуля. Например, предположим, что в модуле `A` определяется функцию с именем `f`, а в модуле `B` импортируется модуль `A` и повторно экспортируется `f`. Теперь, если модуль `C` импортирует модуль `B`, он может ссылаться на `f` по имени `B.f`, хотя первоначально `f` было определено в модуле `A`. Это полезный способ работы с пространством имен; он означает, что библиотеку можно использовать в любом месте внутри структуры модулей, где вы пожелаете, и, при этом, предоставлять понятный и аккуратный интерфейс API через несколько интерфейсных модулей, в которые повторно экспортируют идентификаторы из внутренних модулей.

Однако компилятор для того, чтобы знать, чему соответствует каждое имя в исходном коде, должен во всем этом разобраться. Мы строго различаем *сущности* (*entities*), т. е. «сами предметы» (в нашем примере, `A.f`), и имена, с помощью которых можно обращаться к сущностям (например, `B.f`). В любом конкретном месте в исходном коде, в области видимости есть набор сущностей, и для каждой из них может быть известно одно или несколько различных имен. Задача блока переименования заключается в замене каждого из имен в внутреннем представлении кода, имеющегося в компиляторе, на ссылку на конкретную сущность. Иногда имя может ссылаться на несколько сущностей; само по себе это не является ошибкой, но если имя действительно используется, то блок переименования установит флаг ошибки неоднозначности и отвергнет программу.

При переименовании в качестве входа берется абстрактный синтаксис Haskell (`HsSyn`, `RdrName`), а на выходе получают (`HsSyn` `Name`). Здесь `Name` является ссылкой на конкретную сущность.

Разрешение имен является основной работой блока переименования, но он также выполняет множество других задач: анализ функций и установка флага ошибки в случае, если у функций не совпадает количество аргументов; преобразование инфиксных выражений согласно приоритету выполнения операций; выявление объявлений-дубликатов; генерация предупреждений о неиспользуемых идентификаторах, и так далее.

Проверка типов

Проверкой типов, как можно было бы предположить, является процесс проверки того, что программа на языке Haskell является корректной с точки зрения используемых типов. Если программа проходит проверку типов, то ее работа гарантированно не закончится крахом во время выполнения. Термин «крах» здесь имеет формальное определение, которое включает в себя сбой работы аппаратных средств, например, «ошибка сегментации», но не включает в себя такие ситуации, как ошибки сопоставления с образцом. Гарантия того, что работа программы не закончится крахом, может быть нарушена использованием определенных небезопасных функций языка, например, использованием интерфейса доступа к внешним функциям Foreign Function Interface.

В качестве входных данных блока проверки типа берется `HsSyn Name` (исходный код на языке Haskell с полностью квалифицированными именами), а на выходе получают `HsSyn Id`. Идентификатор `Id` является именем `Name` с дополнительной информацией: в частности *типом*. На самом де-

ле, синтаксические конструкции языка Haskell, создаваемые блоком проверки типов, сопровождаются полным набором информации, касающейся типов: для каждого идентификатора указывается его собственный тип данных, а также есть вся информация, достаточная для преобразования подвыражений любых типов (например, это может использоваться в оболочках IDE).

На практике, проверка типов и переименование могут чередоваться, т.к. из-за использованию шаблонов языка Haskell во время выполнения происходит генерация кода, для которого необходимы переименование и проверка типов.

Сокращение синтаксического разнообразия языка и язык Core

Язык Haskell является довольно большим языком, в котором есть много различных синтаксических форм. Его назначение быть легким для тех, кто на нем читает и пишет; в нем присутствует широкий спектр синтаксических конструкций, которые дают программисту большую гибкость в выборе наиболее подходящей конструкцией для конкретной ситуации. Однако эта гибкость означает, что часто для одного и того же кода есть несколько способов записи; например, выражение `if` по смыслу идентично выражению `case` с вариантами `True` и `False`, а сложная нотация списков может быть преобразована в обращения к функциям `map`, `filter` и `concat`. В действительности, в определении языка Haskell указывается, как все эти конструкции переводятся в более простые конструкций; конструкций, в которые можно выполнить преобразование без всякого, так называемого, «синтаксического сахара».

Для компилятора гораздо проще, если весь синтаксический сахар удален, поскольку при последующих проходах оптимизации, которые необходимо использовать с программой на языке Haskell, работа будет происходить с языком меньшего размера. Поэтому процесс сокращения синтаксического разнообразия языка удаляет весь синтаксический сахар, переводя полный синтаксис языка Haskell в гораздо меньший язык, который мы называем базовым языком. Мы позже поговорим подробно о базовом языке Core.

Оптимизация

Теперь, когда программа представлена в виде базового языка Core, начинается процесс оптимизации. Одна из самых сильных сторон компилятора GHC состоит в оптимизации, в ходе которой убираются слои абстракции и вся эта работа происходит на уровне базового языка Core. Язык Core является крошечным функциональным языком, но это чрезвычайно гибкое средство для выполнения оптимизаций, начиная с самого высокого уровня, например, проверки строгости конструкций, и до самого низкого уровня, например, сокращения числа действий.

Each of the optimisation passes takes Core and produces Core. The main pass here is called the Simplifier, whose job it is to perform a large collection of correctness-preserving transformations, with the goal of producing a more efficient program. Some of these transformations are simple and obvious, such as eliminating dead code or reducing a case expression when the value being scrutinised is known, and some are more involved, such as function inlining and applying rewrite rules (discussed later). При каждой оптимизации берется язык Core и создается язык Core. При этом главный проход называется *упрощающим* (*Simplifier*), работа которого заключается в применении большого набора преобразований, не нарушающих правильность программы, с целью получения более эффективной программы. Некоторые из этих преобразований просты и очевидны, например, устранение недоступного кода или сокращение выражения case после тщательного исследования значения, а некоторые — более сложные, например, подстановка функций непосредственно в код и применение правил грамматического вывода (будут рассмотрены ниже).

Упрощающий подход обычно выполняется между другими проходами оптимизации, которых насчитывается до шести; какие проходы на самом деле работают и в каком порядке, зависит от уровня оптимизации, который выбирает пользователь.

Генерация кода

После того, как будет оптимизирована программа `Core`, начинается процесс генерации кода. После нескольких административных проходов, код преобразуется программе одного из следующих двух видов: либо он превращается в *байт-код* для исполнения интерактивным интерпретатором, либо он передается в *генератор кода* для последующего перевода в машинный код.

Генератор кода сначала преобразует программу `Core` в язык, называемый `STG`, который, по существу, представляет собой аннотированный вариант `Core` с дополнительной информацией, которая необходима генератору кода. Затем `STG` транслируется в `Cmm`, низкоуровневый императивный язык с явным использованием стека. С этого момента преобразование кода может пойти по одному из следующих трех вариантов:

- **Генерация нативного кода:** В компиляторе `GHC` есть простые генераторы кода для нескольких вариантов архитектуры процессоров. Этот вариант выполняется быстро и создается код, который подходит для большинства случаев.
- **Генерация кода LLVM:** `Cmm` преобразуется в код на языке LLVM и передается компилятору LLVM. В этом варианте в некоторых случаях может создаваться значительно лучший код, хотя на это требуется больше времени, чем генератору нативного кода.
- **Генерация кода на языке C:** Компилятор `GHC` может создавать код на обычном языке C. Этот вариант существенно более медленный, чем два предыдущих варианта, но он может быть полезен при портировании компилятора `GHC` на новые платформы.

5.3. Ключевые проектные решения

В этом разделе мы сосредоточимся на нескольких проектных решений, которые оказались особенно эффективными в компиляторе `GHC`.

Промежуточный язык

Выражения>			
t, e, u	$::=$	x	Переменные
		K	Конструкторы данных
		k	Литералы
		$\lambda x:\sigma.e e\ u$	Абстракция значений и приложения
		$\Lambda a:\eta.e e\ \varphi$	Абстракция типов и приложения
		$\text{let } x:\tau = e \text{ in } u$	Локальное связывание
		$\text{case } e \text{ of } p \rightarrow u$	Выражения типа case
		$e \triangleright \gamma$	Приведения типов
		$\lfloor \gamma \rfloor$	Ограничения
p	$::=$	$K c:\eta x:\tau$	Шаблоны

Рис.5.3: Синтаксис языка `Core`

Типичная структура компилятора для статически-тиปизированного языка следующая: в программе проверяются типы данных и, прежде чем будет выполняться оптимизация, программа будет преобразована в некоторый *промежуточный нетипизированный* язык. Компилятор `GHC` отличается: в нем используется *статически-типизированный промежуточный* язык. Как оказалось, это проектное решение повлияло на проектирование и разработку компилятора `GHC`.

Промежуточный язык, используемый в компиляторе GHC, называется `Core` (когда рассматриваем реализацию) или системой Fc (когда мы имеем дело с теорией). Его синтаксис приведен на рисунке 5.3. Точные детали здесь не важны, более подробную информацию заинтересованный читатель может найти в [SCPD07]. Однако, для наших целей ключевыми являются следующие моменты:

- Язык Haskell является языком с очень большим исходным кодом. Тип данных, представляющий его синтаксическое дерево, имеет буквально сотни конструкторов.

В отличие от него язык `Core` является крошечным языком, являющийся, в принципе, лямбда-исчислением. В нем крайне мало синтаксических форм, но мы можем преобразовывать все конструкции языка Haskell в язык `Core`.

- Язык Haskell является языком с неявно типизированным исходным кодом. В программе может быть мало или совсем нет аннотаций типов; вместо этого в ней используется алгоритм вывода типов для того, чтобы выяснить тип каждого компонента и подвыражения. Этот алгоритм вывода типов сложный и иногда опирается на проектные компромиссы, которые есть в каждом настоящем языке программирования.

Язык `Core`, наоборот, является языком с явной типизацией. Для каждого компонента явно указывается его тип, а в термах (конструктах языка — прим.пер) явно указываются типы и правила использования. В языке `Core` используется очень простой, быстрый алгоритм проверки типов, который проверяет, что программа является правильной с точки зрения использования типов. Алгоритм достаточно очевиден; специальные компромиссы отсутствуют.

Весь анализ и оптимизация, которые выполняются компилятором GHC, переносятся в язык `Core`. Это очень удобно: поскольку `Core` является таким крошечным языком, есть всего несколько вариантов оптимизации. Хотя язык `Core` небольшой, он крайне выразителен, система F, в конце концов, изначально разрабатывалась как фундаментальное исчисление для типизированных расчетов. Когда к языку, на котором пишется исходный код, добавляются новые возможности (что иногда происходит), изменения, как правило, ограничены только внешним языком; неизменным остается язык `Core`, и, следовательно, большая часть компилятора.

Но почему язык `Core` типизирован? В конце концов, если движок вывода типов получает на вход исходную программу, то эта программа, по-видимому, хорошо типизирована и на каждой фазе оптимизации, как предполагается, сохраняется такая корректность типов, но почему вы всегда хотите запускать этот движок? Более того, для того, чтобы сделать язык `Core` типизированным, требуются значительные затраты; поскольку при каждом проходе преобразования или оптимизации требуется создавать правильно типизированную программу, а генерация аннотаций всех таких типов часто является нетривиальной задачей.

Тем не менее, наличие явно типизированного промежуточного языка стало огромной победой по следующим нескольким причинам:

- Запуск программы проверки типов в `Core` (назовем ее `Lint`) является очень мощным средством целостности данных в самом компиляторе. Представьте себе, что вы пишете «оптимизацию», случайно создающую код, который трактует целое значение как функцию, и пытается ее вызвать. Есть вероятность того, что в программе произойдет ошибка сегментации, либо во время выполнения программы причудливым образом возникнет ошибка. Трассировка ошибки сегментации вернет программу обратно на соответствующий этап оптимизации, из-за которой программа была выведена из строя.

А теперь представьте, что вместо этого мы запускаем проверку типов `Lint` после каждого прохода оптимизации (и мы делаем это, если вы используете флаг `-dcore-lint`): она сразу

после неправильной оптимизации точно сообщит о том, где находится ошибка. Хвала всему вышнему.

Конечно, устойчивость типов не то же самое, что и корректность: `Lint` не будет сигнализировать об ошибке, если вы «оптимизировали» выражение ($x * 1$) до 1 вместо x . Но если программа проходит проверку `Lint`, то это будет гарантировать, что она будет работать без ошибок сегментации, и, кроме того, на практике мы обнаружили, что это удивительно трудно случайно написать оптимизацию, при которой сохраняется правильность типов, но нарушаются семантическая корректность.

- Алгоритм вывода типов для языка Haskell очень большой и очень сложный: взгляд на рисунок 5.1 подтверждает, что контроль типов на сегодняшний день является самым большим компонентом компилятора GHC. Большой размер и сложность означают то, что могут быть ошибки. Но `Lint` выступает в роли 100%-й независимой проверки типов в движке вывода типов; если движок вывода типов принимает программу, которая, на самом деле, не является корректной с точки зрения типов, то `Lint` отвергнет ее. Поэтому `Lint` выступает в роли мощного аудитора движка вывода типов.
- Наличие языка `Core` также оказалось важной проверкой на вменяемость конструкций исходного языка. Наши пользователи постоянно предлагают новые функции, которые они хотели бы видеть в языке. Иногда эти функции являются явным «синтаксическим сахаром», т. е. удобным новым синтаксисом для чего-то, что вы можете уже делать. Но иногда они могут более глубокими, и, возможно, сразу трудно сказать, как далеко будет простираться эта функция.

Язык `Core` дает нам точный способ оценить такие возможности. Если функция может быть легко преобразована в `Core`, то это убеждает нас в том, что ничего принципиально нового не происходит: новая функция является лишь новым синтаксическим представлением. С другой стороны, если для этого потребуется расширение языка `Core`, то мы рассматриваем предложение намного более тщательно.

На практике язык `Core` оказался невероятно стабильным: в течение 20-летнего периода мы добавили в `Core` ровно одну новую основную возможность (а именно, `coercions` и связанные с ней приведения типов). За тот же период, исходный язык был расширен весьма существенно. Мы объясняем эту стабильность не нашими собственными достижениями, а, скорее всего, тем, что `Core` базируется на фундаментальной математике: браво Жерар!

Проверка типов исходного языка

Одно интересное проектное решение связано с тем, должна ли проверка типов осуществляться до или после приведения исходного языка к языку `Core`. Компромиссы, таковы:

- Проверка типов до выполнения приведения означает, что программа проверки типов должна иметь дело непосредственно с очень большим синтаксисом языка Haskell, поэтому программа проверки типов должна рассматривать большое количество вариантов. Если мы сначала преобразуем исходный вариант в (бестиповый вариант) `Core`, то можно надеяться, что программа проверки типов станет гораздо меньше.
- С другой стороны, проверка типов после выполнения приведения налагает значительные новые обязательства: приведение, которое не влияет ни на какие программы, является корректным с точки зрения типов. В конце концов, приведение подразумевает преднамеренную потерю информации. Вероятно, что в 95% случаев никаких проблем не будет, но любая возникающая здесь проблема требует некоторого компромисса в разработке `Core` с тем, чтобы сохранить некоторую дополнительную информацию.

- Самым серьезным является то, что проверка типов в программе после приведения очень сильно усложняет выдачу сообщения об ошибках со ссылкой на оригиналный текст программы, а не на его (иногда сложную) версию, полученную после приведения программы.

Большинство компиляторов выполняет проверку типов после приведения программы, но для компилятора GHC мы сделали иной выбор: мы делаем проверку типов для полного синтаксиса исходного языка Haskell, а затем для полученного результата осуществляем приведение. Кажется, что добавлять новые синтаксические конструкции станет сложно, но (следуя французской школе разработки компиляторов) мы структурировали движок вывода типов таким образом, что это делается достаточно просто. Вывод типов состоит из двух частей:

1. Генерация ограничений: выполняется обход синтаксического дерева исходного кода и генерируется коллекция ограничений типов. На этом шаге рассматривается полный синтаксис языка Haskell, но это очень простой код и в нем легко добавлять новые случаи.
2. Решения, относящиеся к ограничениям: принимаются решения относительно каждого созданного ограничения. Это именно то место, где используется движок вывода типов, но он не зависит от синтаксиса исходного языка и был бы точно такой же для существенно меньшего или гораздо большего языка.

В целом, выбор варианта, когда проверка типов осуществляется перед приведением программы, дает больший выигрыш. Да, при этом увеличивается количество строк кода в программе проверки типов, но это *простые* строки. Исчезает ситуация, когда одному и тому же типу данных присваивается две противоречивые роли, и движок вывода типов становится менее сложным и его легче изменять. Кроме того, сообщения компилятора GHC об ошибке типов становятся сравнительно простыми.

Без таблиц символов

В компиляторах обычно есть одна или большее количество структур данных, известных как *таблицы символов*, которые являются отображением символов (например, переменных) в некоторую информацию о переменной, например, информацию о ее типе, или о месте, где она была определена в исходном коде.

В компиляторе GHC мы используем таблицы символов довольно скромно — главным образом при переименовании и проверке типов. Там, где это возможно, мы используем альтернативные стратегии: переменная является структурой данных, в которой содержится вся информация о самой переменной. Действительно, когда мы проходим через структуру данных переменной, то можем получить большое количество информации: из переменной мы можем узнать ее тип, в котором есть конструкторы типа, которые содержат конструкторы данных, в которых самих есть типы данных, и так далее. Например, вот некоторые типы данных компилятора GHC (сильно сокращенный и упрощенный вариант):

```
data Id      = MkId Name Type
data Type    = TyConApp TyCon [Type]
             | ...
data TyCon   = AlgTyCon Name [DataCon]
             | ...
data DataCon = MkDataCon Name Type ...
```

Идентификатор `Id` содержит свой тип `Type`. Тип `Type` может быть применением конструктора типа к некоторым аргументам (например, `Maybe Int`), в этом случае он содержит тип `TyCon`. Тип `TyCon` может быть алгебраическим типом данных, в этом случае в нем указывается список его конструкторов данных. В каждом конструкторе `DataCon` есть свой собственный тип `Type`, о котором, конечно, упоминается в типе `TyCon`. И так далее. Все составляющие структуры тесно взаимосвязаны.

В самом деле может быть цикл: тип данных `TyCon` может содержать конструктор `DataCon`, в котором есть тип `Type`, в которое содержится тот самый тип `TyCon`, с которого мы начали.

В таком подходе есть ряд преимуществ и недостатков:

- Многие запросы, для которых требуется поиск в таблице символов, сводятся к доступу к простому полю, что очень хорошо для обеспечения эффективности и ясности кода.
- Нет необходимости иметь дополнительные таблицы символов, поскольку в абстрактном синтаксическом дереве уже есть вся информация.
- Меньше накладные расходы по затратам памяти: все экземпляры одной и той же переменной совместно используют одну и ту же структуру данных, и для таблицы не нужно отводить дополнительную память.
- Трудности возникают только в случае, когда нам нужно изменить какую-нибудь информацию, связанную с переменной. Это то, где у таблицы символов есть преимущество: мы просто изменяем запись в таблице символов. В компиляторе GHC мы должны обойти абстрактное синтаксическое дерево и заменить все экземпляры старой переменной на новую; более того, в блоке, с помощью которого происходит упрощение программы, это делается регулярно, поскольку ему необходимо обновлять определенную информацию о каждой переменной, относящейся к вопросам оптимизации.

Трудно определить, будут ли в общем использоваться таблицы символов лучше или хуже, поскольку этот аспект разработки настолько фундаментален, что его почти невозможно изменить. Тем не менее, отказ от использования таблиц символов является естественным выбором в чисто функциональной среде, поэтому вполне вероятно, что такой подход является хорошим вариантом для языка Haskell.

Межмодульная оптимизация

Функциональные языки поощряют программиста писать небольшие определения. Например, определение `&&` из стандартной библиотеки выглядит следующим образом:

```
(&&) :: Bool -> Bool -> Bool
True && True = True
      && _       = False
      _           = False
```

Если при каждом использовании такой функции действительно потребуется вызов функции, то эффективность будет ужасной. Одним из решений является заставить компилятор обрабатывать некоторые функции специальным образом, другое решение состоит в использовании препроцессора, который заменит вызов «call» прямой вставкой кода в строку (*inline code*). Все эти решения неудовлетворительны в той или иной форме, тем более, что настолько очевидно другое решение: просто вставить функцию. «Вставить функцию» означает замену вызова функции на копию тела функции с подстановкой соответствующих экземпляров ее параметров.

В компиляторе GHC мы систематически использовали этот подход [PM02]. В компилятор практически ничего не встроено. Вместо этого, мы для того, чтобы устраниТЬ накладные расходы, как можно больше определяем все в библиотеках и агрессивно используем вставки в код. Это означает, что *программисты могут определять свои собственные библиотеки, которые будут вставляться в код и будут оптимизироваться, а также те библиотеки, которые приходят вместе с компилятором GHC*.

Следствием является то, что в компиляторе GHC должна быть возможность делать в код межмодульные вставки и даже межпакетные вставки. Идея проста:

- Когда компилируется модуль `Lib.hs` на языке Haskell, то компилятор GHC создает объектный код в файле `Lib.o` и «интерфейсный файл» в `Lib.hi`. В этом интерфейсном файле находится информация обо всех функциях, которые экспортируются в `Lib`, в том числе указываются как их типы, так и, если функции достаточно малы, и их определения.
- Когда компилируется модуль `Client.hs`, в котором импортируется библиотека `Lib`, компилятор GHC читает интерфейс `Lib.hi`. Таким образом, если в `Client` вызывается функция `Lib.f`, определенная в `Lib`, то компилятор GHC может использовать информацию, имеющуюся в `Lib.hi`, для того, чтобы вставить в код функцию `Lib.f`.

По умолчанию компилятор GHC будет показывать определение функции в интерфейсном файле только в том случае, если функция «маленькая» (есть флаги, контролирующие размер этого порога). Но мы также поддерживаем параметр `INLINE Pragma`, который указывает компилятору GHC в любом случае вставлять следующим образом определение везде, где используется вызов, причем независимо от его размера:

```
foo :: Int -> Int
{-# INLINE foo #-}
foo x = <некоторое большое выражение>
```

Кросс-модульная вставка в код является абсолютно необходимым средством для получения суперэффективных библиотек, но это имеет свою цену. Если автор обновляет свою библиотеку, то недостаточно перекомпоновать файл `Client.o` с новой библиотекой `Lib.o`, поскольку в `Client.o` есть фрагменты старой библиотеки `Lib.hs`, которые вставлены непосредственно в код и которые могут оказаться несовместимыми с новой библиотекой. Другой способ — это указать, что ABI (Application Binary Interface) библиотеки `Lib.o` был изменен так, что требуется перекомпиляция клиентских модулей.

На самом деле, единственным способом компиляции кода, сгенерированного с фиксированным предсказуемым интерфейсом ABI, является отключение межмодульной оптимизации, а это, как правило, слишком высокая цена, чтобы заплатить за совместимость с ABI. У пользователей, работающих с компилятором GHC обычно есть весь исходный код, поэтому перекомпиляция, как правило, не является проблемой (и, как мы далее узнаем, пакет `system` создан именно для такого режима работы). Тем не менее, есть ситуации, когда с практической точки зрения перекомпиляция не подходит: например, распространение исправлений ошибок в библиотеках, которые поставляются в двоичном дистрибутиве ОС. В будущем, как мы надеемся, возможно удастся найти компромиссное решение, которое позволит сохранить совместимость с ABI и, то же время, позволит использовать некоторые варианты межмодульной оптимизации.

5.4. Средства расширяемости

Часто бывает, что проект живет или умирает в зависимости от того, насколько он расширяем. Монолитный кусок программного обеспечения, которой не является расширяемым, должен делать все и должен делать это правильно, в то время как расширяемый фрагмент программного обеспечения, может быть полезным даже в случае, если в нем не еще реализованы сразу «из коробки» все необходимые функции.

Проекты с открытым исходным кодом, конечно, расширяемые по определению, причем каждый разработчик может взять код и добавить в него свои собственные функции. Но модификация оригинального исходного кода проекта, осуществляемая кем-то еще, не только требует больших накладных расходов, но также не способствует распространению ваших расширений среди других пользователей. Поэтому в успешных проектах, как правило, предоставляются свои собственные варианты расширений, которые не связаны с изменением основного кода, и в этом отношении компилятор GHC не является исключением.

Правила преобразований, определяемые пользователями

Ядро компилятора GHC представляет собой длинную последовательность проходов оптимизации, каждый из которых выполняет некоторое преобразование из Core в Core, сохраняющее семантику. Но автор библиотеки определяет функции, для которых часто требуются некоторые свои собственные нетривиальные предметно-ориентированные преобразования, которые никак не могут быть предсказаны компилятором GHC. Итак компилятор GHC позволяет авторам библиотек определять *правила переписывания* (*rewrite rules*), которые применяются для того, чтобы преобразовывать программы в процессе оптимизации [PTH01]. Таким образом, программисты могут, по сути, расширять компилятор GHC с помощью предметно-ориентированных вариантов оптимизации.

Одним из примеров является правило `foldr/build`, которое выражается следующим образом:

```
{-# RULES "fold/build"
  forall k z (g::forall b. (a->b->b) -> b -> b) .
    foldr k z (build g) = g k z
#-}
```

Все правило является параметром `pragma`, которое вводится с помощью `{-# RULES`. В правиле говорится, что всякий раз, когда компилятор GHC видит выражение `(foldr k z (build g))`, он должен переписать его как `(g k z)`. Это преобразование сохранит семантику, но что это именно так, нужно изучить статью [GLP93], так что нет никаких шансов, что компилятор GHC выполнит это правило автоматически. Если воспользоваться еще несколькими другими правилами и некоторыми параметрами `INLINE pragma`, то компилятор GHC сможет сливать вместе функции преобразования списков. Например, два цикла `(map f (map g xs))` объединяются в один.

Хотя правила переписывания просты и ими легко пользоваться, они оказались очень мощным механизмом расширения. Когда мы десять лет назад впервые добавили эту возможность в компилятор GHC, мы предполагали, что это будет полезная возможность, которой будут пользоваться лишь изредка. Но на практике она оказалась полезной в очень многих библиотеках, эффективность которых часто во многом зависит от правил переписывания. Например, в собственной библиотеки компилятора GHC `base` содержится свыше 100 правил, а в популярной библиотеке `vector` используется несколько десятков правил.

Плагины компилятора

Один из способов, с помощью которого можно расширять компилятор, это разрешить программистам писать проход компиляции, который будет вставлен непосредственно в конвейер, используемый в компиляторе. Такие проходы часто называются «плагинами». В GHC плагины поддерживаются следующим образом:

- Программист, скажем, пишет проход из Core в Core, в виде обычной функции языка Haskell в модуле `P.hs` и компилирует его в объектный код.
- При компиляции модуля программист использует флаг командной строки `-plugin P`. Либо он может в начале модуля задать флаг в параметре `pragma`.
- GHC ищет модуль `P.o`, динамически компонует его с работающим двоичным модулем GHC и вызывает модуль в соответствующем месте в конвейере.

Но что такое «соответствующее место в конвейере»? Компилятор GHC этого не знает, и поэтому позволяет плагину принять соответствующее решение. Из-за этого и ряда других причин, интерфейс API, который должен быть реализован в плагине, несколько сложнее, чем просто функция, действующая из Core в Core, но не более.

Для плагинов иногда требуются вспомогательные данные, либо плагины сами могут создавать такие данные. Например, плагин может выполнить некоторый анализ функций в компилируемом модуле (скажем, в модуле `M.hs`), и, возможно, есть смысл поместить эту информацию в интерфейсный файл `M.hi`, с тем чтобы у плагина был доступ к этой информации в случае, когда компилируются модули, в которые импортируется модуль `M`. Для поддержки такой возможности в компиляторе GHC предоставляется механизм аннотаций.

Плагины и аннотации являются относительно новыми возможностями в компиляторе GHC. Они имеют более высокий порог входления, чем правила переписывания, поскольку плагин работает с внутренними структурами данных компилятора GHC, но с их помощью, конечно, можно делать гораздо больше. Еще предстоит выяснить, насколько широко будут они использоваться.

Компилятор GHC как библиотека: интерфейс API компилятора GHC

Одной из первоначальных целей разработки компилятора GHC было создание *модульной* базы, на основе которой можно было бы строить все остальное. Мы хотели, чтобы код GHC был максимально прозрачным и настолько хорошо документированным, насколько это было возможным, для того, чтобы его могли использовать другие разработчики в качестве основы для исследовательских проектов; нам казалось, что другие разработчики захотели бы внести в компиляторе GHC свои собственные изменения с тем, чтобы добавить новые экспериментальные возможности или варианты оптимизации. Действительно, было несколько примеров таких изменений: например, есть версия GHC с языком Lisp в качестве входного интерфейса, а также версия GHC, которая генерирует код на языке Java, причем обе разработки делались совершенно независимо лицами, которые очень слабо контактировали или совсем не контактировали с командой разработчиков компилятора GHC.

Однако создание модифицированных версий компилятора GHC представляет собой лишь небольшую часть из тех приложений, в которых можно повторно использовать код компилятора GHC. По мере того, как растет популярность языка Haskell, наблюдается увеличение потребности в инструментальных средствах и инфраструктуре, которые должны уметь анализировать исходный код на языке Haskell, и в компиляторе GHC, конечно, есть много функций, необходимых для создания подобных инструментальных средств: парсер (синтаксический анализатор) языка Haskell, абстрактный синтаксис, средства проверки типов и так далее.

Имея это в виду, мы внесли в GHC простое изменение: вместо сборки GHC в виде монолитной программы, мы собираем GHC в виде *библиотеки*, которая компонуется затем с небольшим модулем *Main*, позволяющим сделать компилятор GHC исполняемым файлом, но которая также поставляется и как библиотека, так что пользователи могут вызывать компилятор из своих собственных программ. Одновременно мы собираем интерфейс API, с помощью которого функциональные возможности компилятора GHC становятся доступными для клиентских модулей. Интерфейс API предоставляет достаточно много функций с тем, чтобы можно было реализовать пакетный компилятор GHC и интерактивную среду GHCI, он позволяет получить доступ к отдельным проходам компиляции, например, синтаксическому анализу и к проверке типов, а также дает возможность выполнять проверку структур данных, создаваемых с помощью этих проходов. Такое изменение позволило разработать широкий спектр инструментальных средств, созданных с применением интерфейса API компилятора GHC, в том числе:

- Инструмент документирования, Haddock, который читает исходный код на языке Haskell и создает документацию на языке HTML.
- Новые версии интерактивной среды доступа GHCI с дополнительными функциями, например, функцией `ghci-haskelline`, которую впоследствии снова вернули в GHC.
- Различные варианты среды разработки IDE, в которыхлагаются расширенные возможности навигации по исходному коду языка Haskell, например, Leksah.

- Простой вариант API - `hint`, с помощью которого можно на лету оценивать исходный код на языке Haskell.

Система пакетов

Система пакетов была в последние годы ключевым фактором роста использования языка Haskell. Ее основное назначение заключается в предоставлении программистам, использующим язык Haskell, возможности пользоваться кодом, созданным другими, поскольку это важный аспект расширяемости: система пакетов применяется в совместно используемом коде самого компилятора GHC и за его пределами.

В системе пакетов реализованы различные фрагменты общей инфраструктуры, которые вместе позволяют достаточно просто пользоваться общим кодом. Благодаря наличию системы пакетов, само сообщество создало очень много совместно используемого кода; вместо того, чтобы полагаться на библиотеки, получаемые из одного источника, программисты, использующие язык Haskell, пользуются библиотеками, созданными всем сообществом. Такая модель хорошо зарекомендовала себя для других языков; например, система CPAN для Perl, хотя то, что язык Haskell является преимущественно компилируемым, а не интерпретируемым языком, ведет к несколько другому набору проблем.

В своей основе система пакетов позволяет пользователю управлять библиотеками кода Haskell, написанными другими людьми, и использовать их в своих собственных программах и библиотеках. Установка библиотеки Haskell выполняется просто с помощью всего одной команды, например, команда:

```
$ cabal install zlib
```

загружает код пакета `zlib` с сайта <http://hackage.haskell.org>, компилирует его с помощью компилятора GHC, устанавливает скомпилированный код где-нибудь в вашей системе (например, в вашем домашнем каталоге в системе Unix), и с помощью компилятора GHC регистрирует установленный пакет. Кроме того, если пакет `zlib` зависит от других пакетов, которые еще не установлены, то перед компиляцией самого пакета `zlib` они также будут загружены, скомпилированы и установлены. Это чрезвычайно удобный способ работы с совместно используемыми библиотеками кода на языке Haskell.

Пакет система состоит из четырех компонентов, причем только первый из них собственно является частью проекта GHC:

- Инструментальные средства для управления *базой данных* пакетов, которая является просто хранилищем информации о пакетах, установленных в вашей системе. Компилятор GHC при запуске читает базу данных пакетов, поэтому ему известно, какие пакеты доступны и где их найти.
- Библиотека, называющаяся `Cabal` (Common Architecture for Building Applications and Libraries - Общая архитектура для сборки приложений и библиотек), в которой реализованы функции сборки, установки и регистрации отдельных пакетов.
- Сайт <http://hackage.haskell.org>, на котором хранятся пакеты, создаваемые и загружаемые на сайт пользователями. Сайт автоматически собирает документацию для пакетов; документацию можно просматривать в режиме online. На момент написания данной статьи, на сайте Hackage было размещено более 3000 пакетов самого различного функционального назначения, в том числе библиотеки баз данных, веб-фреймворки, инструментальные средства для создания графических интерфейсов, программы для различных структур данных и работы с сетями.
- Инструментальное средство `cabal`, который связывает воедино сайт Hackage и библиотеку Cabal: с его помощью осуществляется скачивание пакетов с сайта Hackage в правильной

последовательности, разрешаются зависимости пакетов и выполняется их сборка, а затем пакеты устанавливаются. На сайт Hackage можно также загружать новые пакеты с помощью команды `cabal`, выполняемой из командной строки.

Эти компоненты разрабатывались в течение нескольких лет членами сообщества Haskell и командой разработчиков компилятора GHC, и теперь эти компоненты вместе образуют систему, которая прекрасно вписывается в модель разработки с использованием открытого исходного кода. Нет никаких барьеров, которые бы мешали совместно использовать код, а также тот код, которые другие разработчики объявляют как совместно используемый (конечно при условии, что вы соблюдаете соответствующие лицензии). Вы можете использовать пакет, который кто-то написал, буквально через несколько секунд после того, как найдете его на сайте Hackage.

Сайт Hackage оказался настолько успешным, что среди оставшихся проблем у него теперь те, что связаны с масштабированием: например, пользователи считают, что трудно делать выбор одного из четырех различных фреймворков баз данных. Текущие разработки направлены на решение этих проблем так, чтобы это устроило сообщество. Например, возможность, разрешающая пользователям комментировать пакеты и назначать им баллы, позволит облегчить поиск лучших и самых популярных пакетов, а возможность, позволяющая собирать от пользователей данные об успешной или неудачной сборке пакетов и получать отчеты, поможет им избегать использовать те пакеты, у которых нет сопровождения или в которых есть проблемы.

5.5. Система времени выполнения

Система времени выполнения (Runtime System - RTS) представляет собой библиотеку, написанную, в основном в коде на C, которая компонуется с каждой программой Haskell. С ее помощью поддерживается инфраструктура, необходимая для запуска скомпилированного кода Haskell. Поддерживаются следующие основные компоненты:

- Управление памятью, в том числе параллельное использование памяти, выделение памяти и работа сборщика мусора;
- управление потоками и планирование их работы;
- Примитивные операции, предоставляемые компилятором GHC;
- Интерпретатор байт-кода и динамический компоновщик для интерактивной среды GHCi.

Оставшаяся часть раздела разделена на две части: во-первых, мы сосредоточимся на нескольких проектных решениях системы RTS, которые, как мы считаем, оказались успешными и благодаря им работа этой системы оказалась настолько хорошей, а, во-вторых, мы поговорим о практике кодирования и инфраструктуре, которую мы создали в системе RTS для того, чтобы справляться с довольно недружелюбной средой программирования.

Ключевые проектные решения

В этом разделе мы рассмотрим два проектных решения системы RTS, которые мы рассматриваем, как особенно успешные.

Слой блоков памяти

Сборщик мусора построен поверх слоя, управляющего блоками памяти, размер которых кратен 4 КБ. Слой блоков памяти имеет очень простой интерфейс API:

```
typedef struct bdescr_ {
    void *                  start;
    struct bdescr_ *         link;
    struct generation_ *    gen;    // генерация
```

```

// .. другие различные поля
} bdescr;

bdescr * allocGroup (int n);
void    freeGroup  (bdescr *p);
bdescr * Bdescr     (void *p); // макро

```

Это единственный интерфейс API, используемый сборщиком мусора для выделения и освобождения памяти. Блоки памяти выделяются с помощью операции `allocGroup` и освобождаются с помощью операции `freeGroup`. В каждом блоке есть небольшая структура, ассоциированная с этим блоком, которая называется *дескриптором блока - block descriptor* (`bdescr`). Операция `Bdescr (p)` возвращает дескриптор блока, ассоциированный с произвольным адресом `p`; это только расчет адреса, который вычисляется исходя из значения `p` и компилируется в несколько арифметических инструкций и манипуляций с битами.

Сборщик мусора должен управлять несколькими различными областями памяти, например, выделяемые по разные нужды, и каждая из этих областей, возможно, должна с течением времени увеличиваться или уменьшаться. Когда области памяти представлены в виде списка связанных блоков, то в сборщике мусора GC не возникает трудностей с выделением в непрерывном адресном пространстве областей памяти различного размера.

В реализации слоя блоков памяти использовались методы, хорошо известные в интерфейсе API языка C — `malloc () / free ()`; поддерживаются списки свободных блоков различных размеров и выполняется объединение соседних свободных областей. Тщательно разрабатывались операции `freeGroup ()` и `allocGroup ()` с тем, чтобы их сложность была $O(1)$.

Одно из главных преимуществ такого проектного решения состоит в том, что для него требуется очень незначительная поддержка со стороны операционной системы, и, следовательно, оно хорошо подходит для обеспечения переносимости. В слое блоков необходимо выделять память размером, кратным 1 Мб, с выравниванием по границе 1 Мб. Хотя ни в каких из обычных ОС такая функциональная возможность не предоставляется непосредственно, она, с учетом того, что эти ОС действительно предоставляют, реализуется без особых трудностей. Преимущество в том, что компилятор GHC не зависит от конкретных особенностей организации адресного пространства, используемого ОС, и он мирно сосуществует с другими потребителями адресного пространства, такими как совместно используемые библиотеки и потоки операционной системы.

Имеются дополнительные накладные расходы, затрачиваемые слоем блоков памяти на управление цепочками блоков вместо того, чтобы пользоваться непрерывно выделенной памятью. Впрочем, мы обнаружили, что эти расходы больше, чем затраты на обеспечение гибкости и переносимости; например, в слоек блоков есть вполне определенный простой алгоритм параллельной сборки мусора, который необходимо реализовать [MHJP08].

Легковесные потоки и распараллеливание

Мы считаем, что параллельная обработка должна быть жизненно важной абстракцией программирования, в частности, для создания таких приложений, как веб-серверы, которые должны одновременно взаимодействовать с большим количеством внешних агентов. Если параллельная обработка является важной абстракцией, то она не должна быть настолько дорогой, что программисты вынуждены ее избегать или создавать сложную инфраструктуру, помогающую справиться с этой сложностью (например, пулы потоков). Мы считаем, что параллельная обработка должна просто работать и должна быть достаточно дешевой с тем, что вы в небольших задачах не беспокоились об образовании новых потоков.

Во всех операционных системах предоставляются потоки, которые работают отлично, проблема в том, что они слишком дороги. Типичная ОС справляется с обработкой тысячи потоков, тогда как мы хотим управлять миллионами потоков.

«Зеленые» потоки (green threads), иначе известные как легковесные потоки или потоки пользовательского пространства, представляют собой хорошо известную технологию, которая позволяет избежать перегрузки потоками операционной системы. Идея состоит в том, что управление потоками осуществляется с помощью самой программы или с помощью библиотеки (в нашем случае, системы RTS), а не с помощью операционной системы. Управление потоками в пространстве пользователя должно быть менее затратным, поскольку в операционной системе будет меньшее количество взаимодействий между потоками.

В системе RTS компилятора GHC мы в полной мере воспользовались этой идеей. Переключение контекста происходит только тогда, когда поток находится в *безопасной точке* (*safe point*), в которой требуется сохранять очень небольшое количество дополнительных состояний. Потому что мы используем аккуратный сборщик мусора с тем, чтобы по требованию можно было перемещать стек потоков и увеличивать или уменьшать его размеры. Эти потоки резко отличаются от потоков операционной системы, в которых при каждом переключении контекста необходимо сохранять все состояние процессора и в которых стеки представляют собой неперемещаемый большой кусок адресного пространства, которое для каждого потока должно быть предварительно зарезервировано.

Зеленые потоки могут быть гораздо более эффективными, чем потоки ОС, так зачем кому-то использовать потоки ОС? С зелеными потоками возникают следующие три основные проблемы:

- Блокировки и внешние обращения. Поток должен иметь возможность обращаться к интерфейсу API операционной системы или к внешней библиотеке, которая блокируется, и не должен блокировать все остальные потоки в системе.
- Распараллеливание. Потоки должны автоматически запускаться параллельно в случае, если в системе есть несколько ядер процессора.
- В некоторых внешних библиотеках (в частности, в OpenGL и в некоторых библиотеках графического пользовательского интерфейса) есть интерфейс API, который должен каждый раз вызываться из одного и того же потока ОС, поскольку в них используется состояние, локальное для конкретного потока.

Оказывается, что все это трудно реализовывать с зелеными потоками. Тем не менее, мы упорно продолжали в компиляторе GHC заниматься зелеными потоками и нашли решения для всех трех типов проблем:

- Когда поток языка Haskell осуществляет внешний вызов, еще один поток операционной системы берет на себя выполнение остальных потоков языка Haskell [MPT04]. Для этой цели поддерживается небольшой пул потоков операционной системы, и новые потоки создаются по требованию.
- Планировщик компилятора GHC мультиплексирует большое количество легковесных потоков языка Haskell в несколько тяжеловесных потоков операционной системы; он реализует прозрачную модель потоков типа M:N. Обычно значение N выбирается равным количеству ядер процессора в машине, что позволяет добиться реального распараллеливания, но без дополнительных накладных расходов, которые есть в случае, когда полновесный поток операционной выделяется для каждого легковесного потока языка Haskell.

Для того чтобы запустить код на языке Haskell, в потоке операционной системы должен быть подготовлена определенная среда (в оригинале статьи она называется *Capability*, т. е. *Способность* или *Возможность* — прим.пер.): структура данных, в которых хранятся ресурсы, необходимые для выполнения кода на языке Haskell, такие как песочница (память,

где создаются новые объекты). В каждый конкретный момент каждая конкретная копия среды может храниться только в одном потоке операционной системы. Мы также называем такую среду «Контекстом выполнения языка Haskell», но в коде в настоящее время используется термин *Capability*.

- Мы предоставляем интерфейс API для создания *граничного потока* (*bound thread*): потока языка Haskell, который привязан к одному конкретному потоку операционной системы, так что любой внешний вызов, осуществляемый потоком языка Haskell, гарантированно будет реализовываться с помощью этого потока операционной системы.

Таким образом, в подавляющем большинстве случаев, потоки языка Haskell ведут себя так же, как потоки операционной системы: в них могут возникать блокировки, связанные с вызовами операционной системы, которые не будут влиять на другие потоки, и они работают параллельно на многоядерной машине. Но они на порядки более эффективны с точки зрения затрат времени и памяти.

При этом в реализации есть одна проблема, с которой пользователи иногда сталкиваются, в частности, в случае, когда измеряют производительность программы. Выше мы говорили, что легковесные потоки выигрывают по эффективности только при переключении контекста в «безопасных точках», местах в коде, которые компилятор определяет как безопасные, когда внутреннее состояние виртуальной машины (стек, куча, регистры и т.д.) находятся в надлежащем порядке и когда может использоваться сборка мусора. В компиляторе GHC безопасной точкой является место, где происходит выделение памяти, что почти во всех программах Haskell происходит достаточно регулярно, так что программы выполняют не более, чем несколько десятков инструкций, не попав в безопасную точку. Тем не менее, в хорошо оптимизированном коде можно найти циклы, которые выполняют много итераций без выделения памяти. Это, как правило, часто происходит в программах оценки производительности (например, в функциях, вычисляющих факториал или числа Фибоначчи). В реальном коде такая ситуация встречается реже, хотя это также случается. Отсутствие безопасных точек не позволяет запустить планировщик, что может иметь пагубные последствия. Этую проблему можно решить, но не без ухудшения производительности таких циклов, и часто это сводится к сохранению результата каждой итерации во внутренних циклах. Возможно, это просто компромисс, с которым мы должны смириться.

5.6. Разработка компилятора GHC

Компилятор GHC является отдельным проектом с двадцатилетним сроком существования, при этом он все еще находится в состоянии постоянных инноваций и разработки. По большей части наша инфраструктура разработки и инструментальные средства являются обычными. Например, мы используем треккер ошибок (Trac), вики (также Trac) и Git в качестве средства контроля версий. Такой контроль версий сначала осуществлялся чисто вручную, затем - CVS, затем - Darcs, пока, наконец, в 2010 году мы не перешли на Git. Есть несколько моментов, которые, возможно, менее обычны, и мы их здесь рассмотрим.

Комментарии и замечания

Одна из наиболее серьезных трудностей в большом долгоживущем проекте — это поддержка технической документации в актуальном состоянии. У нас нет серебряной пули, но мы предлагаем один малотехнологичный прием, который послужил нам особенно хорошо: замечания Notes.

При написании кода, часто возникает момент, когда осторожный программист захочет мысленно сказать что-то вроде следующего: «У этого типа данных есть важный инвариант». Он сталкивается с двумя вариантами, которые оба ему не нравятся. Он может добавить инвариант как комментарий, но из-за этого определение типа может стать слишком длинным так, что трудно будет понять,

что является конструкторами. Кроме того, он может документировать инвариант в другом месте, и есть риск, что данные устареют. За двадцать лет *устареет все!*

Исходя из такой мотивации мы разработали следующее очень простое соглашение:

- Комментарии каких-либо значительных размеров не интегрируются в код, а вместо этого они записываются отдельно с заголовком стандартного вида следующим образом:
- Note [Equality-constrained types]
- ~~~~~
- The type forall ab. (a ~ [b]) => blah
- is encoded like this:
-
- ForAllTy (a:*) \$ ForAllTy (b:*) \$
- FunTy (TyConApp (~) [a, [b]]) \$
- blah
- В то место, к которому имеет отношение комментарий, мы добавляем краткий комментарий со ссылкой на замечание:
- data Type
- = FunTy Type Type -- See Note [Equality-constrained types]
-
- | ...

Комментарий указывает, что имеется нечто интересное, и дает точную ссылку на комментарий, в котором это объясняется. Звучит тривиально, но точность выше по сравнению с нашей предыдущей привычкой указывать «смотрите комментарий выше», поскольку часто не ясно, о каком из многочисленных комментариев, указанных выше, ведется речь, причем через несколько лет комментарий может оказаться даже не выше (он может быть ниже или совсем исчезнуть).

Мало того, что можно перейти от кода, в котором указывается ссылка на замечание, к самому замечанию, возможно также и обратное, что тоже часто бывает полезным. Кроме того, на одно и тоже замечание можно ссылаться из нескольких мест в коде.

Это простая методика, использующая только код ASCII без автоматизированной поддержки, преобразовала нашу жизнь: в компиляторе GHC есть около 800 замечаний и их количество растет с каждым днем.

Как поддерживать рефакторинг

Код компилятора GHC создается так же быстро, как это было десять лет назад, если не быстрее. Нет сомнения в том, что за этот период времени сложность системы возросла многократно; мы ранее рассказывали об общем количестве кода в компиляторе GHC. Тем не менее, система остается управляемой. Мы связываем это с тремя основными факторами:

- Ничего не заменит хорошей инженерии программного обеспечения. Модульность всегда окупается: когда между компонентами создается интерфейс API как можно меньшего размера, то это делает отдельные компоненты более гибкими, потому что у них меньше взаимозависимостей. Например, благодаря типу данных `Core{}` компилятора GHC взаимозависимость между проходами из `Core` в `Core` настолько малой, что отдельные проходы становятся почти полностью независимы и их можно выполнять в произвольном порядке.
- В сравнении с разработка в строго типизированном языке, рефакторинг выглядит просто легким ветерком. Всякий раз, когда нам нужно изменить тип данных или изменить количество аргументов или тип функции, компилятор немедленно сообщает нам, что в других местах в коде нужны исправления. Просто есть абсолютная гарантия того, что статически будет исправлен большой класс ошибок и будет сэкономлено огромное количество време-

ни, и, в особенности, при выполнении рефакторинга. Страшно представить, какое количество вручную написанных тестов нам потребовалось для того, чтобы обеспечить такой же уровень покрытия кода, который обеспечивает система типов.

- При программировании в чисто функциональном языке, трудно представить случайные зависимости, связанные с состояниями. Если вы решите, что вам вдруг понадобится доступ к некоторому состоянию глубоко в алгоритме, то в императивном языке вы могли бы испытать желание просто сделать это состояние глобально видимым, а не явно передавать его туда, где в нем есть необходимость. Такой способ, в конечном итоге, приводит к сплетению невидимым зависимостей и к *хрупкому коду*: коду, который легко можно испортить при модификации. Чисто функциональное программирование заставляет вас делать все зависимости явными, что оказывает некоторое негативное давление на добавление новых зависимостей, а меньшее количество зависимостей означает большую модульность. Конечно, когда *необходимо* добавить новую зависимость, то для того, чтобы выразить зависимость, чистота подхода заставит вас написать больший объем кода, но, на наш взгляд, это та цена, которую стоит заплатить за долгосрочное качество кода.

А в качестве дополнительного преимущества, чисто функциональный код является при сборке потокобезопасным и его, как правило, легче распараллеливать.

Отступления от правил

Когда мы оглядываемся назад на те изменения, которые нам потребовалось сделать для того, чтобы проект GHC вырос таким образом, нам становится очевиден общий урок: когда мы отходили от чистой функциональности, будь то в целях эффективности или удобства, то, как правило, это впоследствии приводило к негативные результатам. Этому у нас есть несколько крупных примеров:

- В компиляторе GHC используется несколько структур данных, в которых есть внутренние преобразования. Одной из них является тип `FastString`, в котором присутствует единая глобальная хэш-таблица, а другой структурой является глобальный кэш `NameCache`, с помощью которого гарантируется, что всем внешним именам будут присвоены уникальные номера. Когда мы попытались распараллелить работу компилятора GHC (то есть, сделать, чтобы компилятор GHC на многоядерном процессоре компилировал нескольких модулей параллельно), то эти структуры данных, в которых использовались внутренние преобразования, оказались единственным камнем преткновения. Если бы мы в этих местах не прибегали к внутренним преобразованиям, то распараллеливание компилятора GHC было бы почти тривиальным.

Хотя мы действительно собрали прототип параллельной версии компилятора GHC, на самом деле в настоящее время в компиляторе GHC нет поддержки параллельной компиляции, но это в значительной степени потому, что мы не затратили усилия, необходимые для того, чтобы сделать эти изменяемых структуры данных поточно-ориентированными.

- Поведение компилятора GHC регулируется в значительной степени флагами командной строки. По определению эти флаги командной строки являются неизменными в течение определенного времени работы GHC, поэтому в ранних версиях компилятора GHC мы сделали значения этих флагов доступными в виде констант верхнего уровня. Например, был флаг верхнего уровня `opt_GlasgowExtS`, имеющий тип `Bool`, с помощью которого указывалось, должны ли использоваться определенные расширения языка или нет. Константы верхнего уровня очень удобны, т.к. там, где к ним в коде нужен доступ, их значения не требуется явно передавать в виде аргументов.

Конечно, эти параметры в действительности не являются константами, поскольку при разных запусках компилятора они могут отличаться, и в определении `opt_GlasgowExtS` есть

вызов `unsafePerformIO`, который скрывает побочный эффект. Как бы это не было, но этот трюк обычно считается «достаточно безопасным», поскольку это значение остается постоянным при любом конкретном запуске; например, это не мешает при компиляции использовать оптимизацию.

Однако позже компилятор GHC был преобразован из одномодульного компилятора в компилятор, состоящий из нескольких модулей. В этот момент трюк с использованием для флагов констант верхнего уровня стал неприемлемым, поскольку при компиляции различных модулей флаги могут иметь различные значения. Так что нам пришлось преобразовывать большой объем кода для того, чтобы значения флагов передавать явным образом.

Возможно, вы бы могли возразить, что флаги следует прежде всего обрабатывать как *состояние*, как бы это естественно происходило в императивном языке, который бы обошел эту проблему. В какой-то степени это действительно так, хотя чисто функциональном коде есть ряд других преимуществ, не последним из которых является то, что представление флагов с помощью неизменных данных означает, что получившийся код будет сразу поточно-безопасным и он сразу без изменений будет работать в режиме распараллеливания.

Разработка системы RTS

Система времени выполнения компилятора GHC представляет собой во многих отношениях разительный контраст с самим компилятором. Есть естественные различия, поскольку система времени выполнения написана на языке C, а не на языке Haskell, но есть также и соображения, уникальные для системы RTS, которые ведут к другой философии проектирования:

1. Каждая программа на языке Haskell тратит много времени на выполнение кода в системе RTS: обычно это около 20 - 30%, но характеристики программ на языке Haskell очень сильно варьируются, поэтому также обычными бывают и значения, выходящие из этого диапазона как в большую, так и в меньшую сторону. Если система RTS оптимизирована, то выгоды возрастают многократно, так имеет смысл потратить много времени и усилий на оптимизацию.
2. Система времени выполнения статически компонуется с каждой программой на языке Haskell (в случаях, когда не используется динамическая компоновка), так что есть стимул сделать эту систему поменьше.
3. Ошибки времени выполнения часто непонятны пользователю (например, «ошибка сегментации») и их трудно обойти. Например, ошибки в сборщике мусора, как правило, не привязаны к использованию конкретной особенности языка, но они возникают, когда во время выполнения возникают некоторые сложные комбинации факторов. Кроме того, ошибки такого рода, как правило, являются непостоянными (случаются только при некоторых запусках программы) и они очень чувствительны к изменениям (крошечные изменения в программе приводят к исчезновению ошибки). С ошибками в многопоточной версии среды выполнения возникает еще больше проблем. Поэтому есть смысл прилагать дополнительные усилия для того, чтобы не допускать таких ошибок, а также для того, чтобы создать инфраструктуру, в которой такие ошибки будут легче идентифицироваться.

Симптомы ошибок системы RTS часто неотличимы от двух других видов отказов: сбоя оборудования, которые происходят гораздо чаще, чем вы думаете, и неправильного использования небезопасных возможностей языка Haskell, например, интерфейса FFI (Foreign Function Interface — интерфейса внешних функций). Первое, что нужно сделать при диагностике отказов времени выполнения, это исключить эти две указанные причины.

4. Система RTS является низкоуровневым кодом, который работает на машинах с несколькими различными вариантами архитектуры и с различными операционными системами, и она

регулярно портируется на новые машины. Возможность портирования имеет важное значение.

Важным является каждый цикл и каждый байт, и, тем более, их правильность. Кроме того, задачи, выполняемые системой времени выполнения, сложны по своей природе, поэтому с самого начала трудно добиться их правильного выполнения. Согласование всех этих особенностей привело нас к некоторым интересным оборонительным приемам, о которых мы расскажем в следующих разделах.

Борьба со сложностью

Система RTS является сложной средой, которая враждебна программированию. В отличие от компилятора, система RTS практически не является типобезопасной. На самом деле, она еще менее типобезопасная, чем большинство других программ на языке C, т.к. она управляет структурами данных, типизация которых поддерживается на уровне языка Haskell, а не на уровне языка C. Например, в системе RTS не известно, что объект, на который ссылается последняя ячейка, является массивом или другим объектом: эта информация просто не существует на уровне языка C. Более того, процесс компиляции кода Haskell стирает информацию о типах, поэтому даже если мы бы сообщили системе RTS о том, что объект, на который происходит ссылка, является списком, все еще бы не было информации о голове списка. Поэтому в коде системы RTS требуется выполнять большое количество преобразований типов указателей языка C, причем с точки зрения безопасности типов это будет для компилятора C очень слабой подмогой.

Поэтому наше первое оружие в этой битве — *избегать добавлений кода в систему RTS*. Всегда, когда это можно, мы помещаем в систему RTS минимальное количество функций, а остальное пишем в библиотеке на языке Haskell. Это редко когда ухудшает результат; код на языке Haskell гораздо более надежен и более компактен, чем на языке C, а его производительность, как правило, вполне приемлема. Точно нельзя сказать, где нужно проводить границу, хотя во многих случаях это достаточно ясно. Например, несмотря на то, что на языке Haskell теоретически можно реализовать сборку мусора, на практике это сделать чрезвычайно сложно, поскольку язык Haskell не позволяет программисту точно контролировать распределение памяти и т.п., поэтому на практике имеет смысл для такой низкоуровневой задачи перейти на уровень языка C.

Есть много функциональных возможностей, которые не удается (легко) реализовать на языке Haskell, а писать их в системе RTS не хочется. В следующем разделе мы сосредоточимся на одном аспекте управления сложностью и корректностью в системе RTS: на использовании инвариантов.

Инварианты и их проверка

В системе RTS есть много инвариантов. Многие из них являются тривиальными и легко проверяются: например, если указатель на голову очереди равен `NULL`, то указатель на хвост очереди также должен быть равным `NULL`. В коде системы RTS есть масса мест, где можно задать утверждения (*assertion*) для проверки аналогичных условий. Проверка утверждения является способом обнаружения ошибок раньше, чем они могут проявиться; на самом деле, когда добавляется новый инвариант, мы прежде чем писать код, реализующий инвариант, часто сначала добавляем утверждение.

Некоторые из инвариантов выявлять и проверять в процессе выполнения программ достаточно трудно. Один инвариант этого вида, который больше относится к системе RTS, состоит в следующем: *в куче не должно быть «висячих» указателей*.

Висячие указатели добавить достаточно просто, и, как в компиляторе, так и в системе RTS, есть много мест, где может быть нарушен этот инвариант. Генератор кода может сгенерировать код, в котором создаются недопустимые объекты кучи; сборщик мусора, когда он сканирует кучу, может забыть обновить указатели некоторого объекта. На отслеживание ошибок этого вида может потребо-

боваться очень много времени (однако, это является одним из любимых занятий автора статьи!), поскольку к тому времени, когда с программой, в конечном итоге, произойдет авария, процесс выполнения программы может пройти достаточно длинный путь от того места, где первоначально был добавлен висячий указатель. Есть хорошие средства отладки, но они не так хороши при прокрутке программы в направлении, обратном ходу ее исполнения. Однако, в последних версиях отладчика GDB и отладчика Microsoft Visual Studio действительно есть некоторая поддержка прокрутки программы в обратном направлении.

Общий принцип следующий: «если в программе дело идет к аварии, то это должно происходить, насколько это возможно, как можно быстрее, с большим количеством сообщений, которые должны выдаваться как можно чаще». Эта цитата взята из правил кодирования компилятора GHC, первоначально написанными Аластером Рейдом (Alastair Reid), который работал над ранней версией системы RTS.

Проблема в том, что не всегда инвариант отсутствия висячих ссылок можно проверить с помощью утверждения, требующего константного времени. Утверждение, с помощью которого делается эта проверка, должно сделать полный обход кучи! Очевидно, мы не сможем использовать это утверждение каждый раз, когда из кучи выделяется память, или каждый раз, когда сборщик мусора сканирует объект. На самом деле, даже этого было бы недостаточно, поскольку висячие указатели не появятся до тех пор, пока память не будет освобождена после окончания работы сборщика мусора.

Поэтому в отладочном варианте системы RTS есть дополнительный режим, который мы называем *проверкой работоспособности* (*sanity checking*). В процессе проверка работоспособности разрешается использовать все виды ресурсоемких утверждений и мы можем выполнять программу во много раз медленнее. В частности, в процессе проверки работоспособности проходит полное сканирование кучи с целью (среди всего прочего) проверить наличие висячих указателей до и после каждого обращения к сборщику мусора. Первое, что нужно сделать для того, чтобы выяснить причину аварии времени выполнения, это запустить программу с включенным режимом проверки работоспособности; иногда это позволит обнаружить нарушение инварианта задолго до того момента, когда программа фактически выходит из строя.

5.7. Заключение

За последние 20 лет авторы настоящей статьи вложили значительную часть своей жизни в проект GHC и мы весьма гордимся тем, как далеко он продвинулся. Это не единственная реализация языка Haskell, но это единственная реализация, которой регулярно пользуются сотни тысяч людей для того, чтобы выполнять реальную работу. Мы постоянно удивляемся, когда язык Haskell начинают использовать в необычных местах; одним из последних примеров является использование языка Haskell для управления системами в грузовиках-мусоровозах.

Для многих языков Haskell и компилятор GHC являются синонимами: эта цель никогда не планировалась, да и во многих случаях было бы контрпродуктивно иметь только одну реализацию стандарта, но дело в том, что поддержка хорошей реализации языка программирования требует большого объема работы. Мы надеемся, что наши усилия, затраченные в проекте GHC на поддержку стандарта и четкого выделения каждого отдельного расширения языка, сделают возможным появление других реализаций и их интеграцию с системой пакетов и другой инфраструктурой. Конкуренция выгодна для всех!

Мы выражаем глубокую благодарность фирме Microsoft, в частности, за предоставленную нам возможность разрабатывать компилятор GHC как часть наших исследований и распространять его с открытым исходным кодом.

6.Git

6.1. Пара слов о Git

Git позволяет хранить копию данных работы (представленную обычно, но не всегда в форме программного кода), выполняемой взаимодействующими друг с другом участниками проекта с использованием распределенной сети репозиториев. Эта система контроля версий поддерживает распределение процессов разработки отдельных частей проекта, позволяя временно вносить несогласия и в конечном счете объединять данные проекта.

В данной главе мы покажем то, как различные части проекта Git выполняют свои функции для реализации описанных возможностей, а также обозначим отличия этого проекта от других проектов систем контроля версий (VCS).

6.2. Начало развития проекта Git

Для лучшего понимания философии, стоящей за архитектурой проекта Git полезно понять обстоятельства, в которых участники сообщества разработчиков ядра Linux начали работу над проектом Git.

Проект разработки ядра Linux отличался от проектов разработки большинства коммерческих приложений того времени с точки зрения разработчиков, так как в его рамках работало большое количество людей, вносящих изменения в исходный код, причем эти люди значительно отличались по степени вовлеченности в проект и знаниям в области существующей кодовой базы. Разработка ядра в течение долгих лет велась с использованием архивов с исходным кодом и патчей, причем основные члены сообщества разработчиков пытались найти систему контроля версий, удовлетворяющую большей части их требований.

Git является проектом с открытым исходным кодом, который был создан для удовлетворения этих требований и исправления существующих недостатков процесса разработки в 2005 году. В это время кодовая база ядра Linux была распределена между двумя системами контроля версий BitKeeper и CVS в соответствии с предпочтениями различных разработчиков. Система BitKeeper предоставляла отличный взгляд на формирование истории системы контроля версий в отличие от популярных систем контроля версий с открытым исходным кодом того времени.

Через некоторое время создавшая систему BitKeeper компания BitMover объявила о намерении отзывать лицензии у некоторых разработчиков ядра Linux. После этого Linus Torvalds второпях начал разработку системы контроля версий, которая в данный момент носит имя Git. Он начал с разработки набора сценариев для упрощения процесса последовательного применения патчей из электронных писем. Целью разработки этого начального набора сценариев была реализация возможности быстрой отмены изменений исходного кода, причем у разработчика должна была быть возможность модификации кодовой базы для последующего наложения патча вручном режиме с последующей возможностью возобновления автоматического применения патчей.

С самого начала разработки Linus Torvalds обозначил философию проекта Git - она была противоположна философии проекта CVS, к тому же он сформулировал три упрощающих использование системы цели проектирования:

- Реализация возможности распределения процессов разработки по аналогии с тем, как это было реализовано в рамках проекта BitKeeper
- Введение защитных мер, предотвращающих повреждение хранимых системой данных
- Достижение высокой производительности системы

Эти цели проектирования были приняты и поддержаны в некоторой степени, о чем я попытаюсь рассказать в ходе исследования методов использования в рамках Git ациклических ориентированных графов (DAG) для хранения данных, указателей ссылок для рабочих веток, представления

объектной модели и удаленного протокола; и наконец о том, как Git отслеживает объединение древовидных представлений данных.

Несмотря на влияние, оказанное на начальную архитектуру Git проектом BitKeeper, реализация архитектурных решений имела фундаментальные отличия и позволила достичь большей степени распределения и возможности работы исключительно в локальном режиме, что было невозможно при использовании BitKeeper. Распределенная система контроля версий [Monotone](#), разработка которой началась в 2003 году, похоже, была еще одним источником вдохновения, используемым на начальном этапе разработки Git.

Распределенные системы контроля версий позволяют достичь значительной гибкости рабочего процесса обычно ценой простоты. Специфические достоинства распределенной модели:

- Представление возможности совместной работы без подключения к сети с последующим инкрементальным внесением изменений в кодовую базу.
- Представление участнику процесса разработки возможности принимать решение о том, готова ли его/ее работа для всеобщего обозрения.
- Представление участнику процесса разработки возможности осуществления доступа к истории изменения репозитория без подключения к сети.
- Возможность публикации имеющейся работы во множестве репозиториев, причем работа может быть опубликована в различных ветвях и с различной видимой детализацией изменений.

Во время начала работы над проектом Git была начата разработка трех других проектов распределенных систем контроля версий с открытым исходным кодом. (Одна из этих систем под названием Mercurial описывается в Главе 1 книги *"Архитектура приложений с открытым исходным кодом"*.) Все эти распределенные системы контроля версий предоставляют незначительно отличающиеся способы формирования гибких рабочих процессов, которые не могли быть непосредственно реализованы при использовании централизованных систем контроля версий, существовавших до них. Примечание: система Subversion предусматривает возможность использования поддерживаемого различными разработчиками расширения SVK, предназначенного для синхронизации данных между серверами.

На сегодняшний день популярными и активно разрабатываемыми распределенными системами контроля версий с открытым исходным кодом являются Bazaar, Darcs, Fossil, Git, Mercurial и Veracity.

6.3. Архитектура системы контроля версий

Сейчас наступил отличный момент для того, чтобы сделать шаг назад и рассмотреть альтернативные Git реализации систем контроля версий. Понимание их отличий позволит исследовать принятые в ходе разработки Git архитектурные решения.

К системе контроля версий обычно предъявляются три ключевых функциональных требования, а именно:

- Хранение данных
- Отслеживание изменений содержимого (хранение истории изменений, включающей метаданные слияния)
- Распространение данных и истории изменений между разработчиками

Примечание: Третье требование из списка выше не является функциональным требованием для всех систем контроля версий.

Хранение данных

Наиболее часто используемыми архитектурными решениями для хранения данных в мире систем контроля версий являются наборы изменений на основе отличий файлов или представление данных в форме ациклического ориентированного графа (DAG).

Наборы изменений на основе отличий файлов инкапсулируют отличия между двумя версиями данных произвольного типа, дополняя их некоторым количеством метаданных. Представление данных в форме ациклического ориентированного графа подразумевает использование объектов, формирующих иерархию, которая зеркалирует содержимое дерева файловой системы в виде копии добавляемых данных (при этом не изменяющиеся объекты в рамках дерева используются повторно тогда, когда это возможно). Git хранит данные в виде ациклического ориентированного графа, использующего различные типы объектов. В разделе "База данных объектов", расположенному ниже этой главе, описываются различные типы объектов, которые могут формировать ациклические ориентированные графы в репозитории Git.

История операций внесения изменений и слияния

В области отслеживания изменений репозитория в большинстве систем контроля версий используется один из следующих подходов:

- Создание линейного представления истории
- Использование направленного ациклического графа для хранения данных истории

Для хранения данных истории в Git как и раньше используются направленные ациклические графы. Каждая операция изменения данных (commit) создает метаданные, указывающие на ее предшествующие этапы; операция изменения данных в Git может не иметь или иметь множество (теоретически неограниченное количество) родительских операций изменения данных. Например, первая операция изменения данных в репозитории Git не будет иметь родительских операций, а результат трехстороннего слияния ветвей будет иметь три родительских операции.

Другим ключевым отличием между системой Git и системой Subversion с ее линейным представлением истории предыдущих операций является возможность непосредственного выделения ветвей, в рамках которых будет производиться запись истории большинства операций объединения.

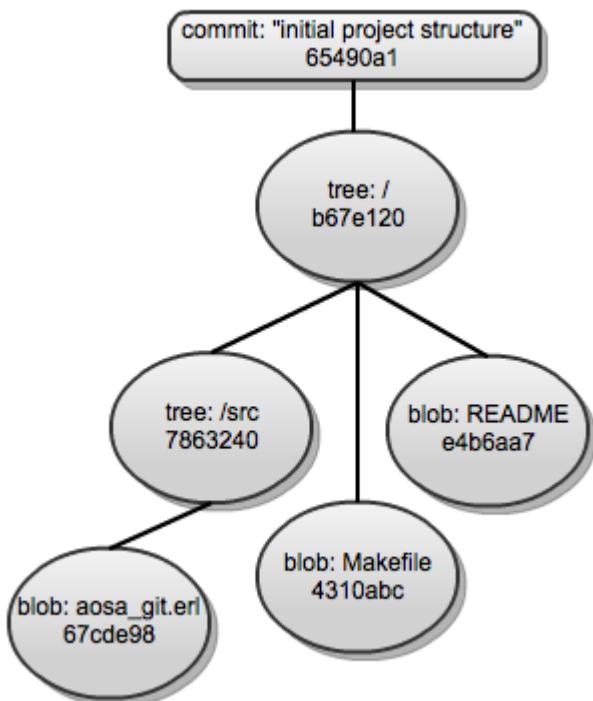


Рисунок 6.1: Пример представления данных истории в форме направленного ациклического графа в Git

Git в полной мере поддерживает возможности выделения ветвей, используя при этом направленные ациклические графы для хранения данных. История изменений файла непосредственно связывается со структурой директорий (с помощью вершин, представляющих директории) до корневой директории, которая впоследствии связывается с ветвью операции изменения данных. Эта ветвь операции изменения данных, в свою очередь, может иметь одного или нескольких родителей. Это обстоятельство наделяет Git двумя возможностями, которые позволяют нам более четко ориентироваться в истории и данных, чем это возможно при использовании семейства систем контроля версий, созданных на основе RCS, а именно:

- В момент, когда узел данных (т.е., файлов или директорий) в графе ссылается на те же данные (использует тот же хэш SHA в Git), что и данные другой операции изменения данных, два узла будут гарантированно содержать идентичные данные, позволяя Git эффективно осуществлять дедупликацию.
- При объединении двух ветвей мы объединяем содержимое двух узлов направленного ациклического графа. Направленный ациклический граф позволяет Git "эффективно" (в сравнении с семейством систем контроля версий на основе RCS) устанавливать стандартные предшествующие узлы.

Распространение данных

Системы контроля версий осуществляют распространение данных рабочей копии между участниками процесса разработки проекта одним из трех способов:

- Исключительно локально: метод применим для систем контроля версий, к которым не предъявляется третьего функционального требования из представленного выше списка.
- С использованием центрального сервера: в данном случае все измененные данные должны передаваться через один определенный репозиторий для занесения данных об их изменениях в историю.
- С использованием распределенной модели: в данном случае репозитории всегда будут открыты для участников процесса разработки, которые смогут "поместить" данные в них, но операции изменения данных могут также осуществляться локально и в этом случае данные будут помещены в репозиторий позднее, что позволяет выполнять работу в условиях отсутствия соединения с сетью.

Для демонстрации преимуществ и недостатков каждого из основных архитектурных решений, мы рассмотрим репозиторий системы Subversion и репозиторий системы Git (на сервере), содержащие идентичные данные (т.е., HEAD-ветвь, указывающая на рабочую ветвь репозитория Git будет содержать те же данные, что и последняя ревизия из поддиректории trunk репозитория Subversion). Разработчик по имени Alex создал локальную копию данных репозитория Subversion и произвел клонирование данных для создания локального репозитория Git.

Представим, что Alex изменяет файл размером в 1 МБ в локальной копии данных из репозитория Subversion, после чего отправляет изменения в репозиторий. Локальная копия отражает последние модификации и локальные метаданные претерпевают изменения. В ходе инициированного Alex процесса добавления данных в централизованный репозиторий Subversion генерируется файл различий между предыдущим экземпляром файла и измененным файлом, после чего этот файл различий сохраняется в репозитории.

В подобной ситуации Git ведет себя диаметрально противоположно. В момент, когда Alex производит точно такую же модификацию эквивалентного файла в локальном клонированном репозитории, изменения сначала будут записаны только локально, после чего Alex сможет "поместить" ожидающие в локальном репозитории изменения в публичный репозиторий, сделав свою работу доступной для других участников процесса разработки проекта. Изменения данных сохраняются идентично в каждом репозитории Git, где они присутствуют. При изменении данных в локальном репозитории (в самом простом случае), локальный репозиторий Git создаст новый объект, представляющий измененный файл (со всеми данными файла внутри). Для каждой директории, находящейся выше измененного файла в дереве директорий (а также для корневой директории репозитория), новый объект дерева будет создан с новым идентификатором. Направленный ациклический граф создается в направлении от недавно созданного объекта корневой директории, указывающего на объекты данных (при этом повторно используются существующие ссылки на объекты данных в том случае, если содержимое файлов не было изменено в ходе данной операции измене-

ния данных), а также указывающего на недавно созданный объект данных в месте, где был расположжен предыдущий объект данных в предыдущей иерархии. (Объект данных (*blob*) представляет файл, хранимый в репозитории.)

В этот момент измененные данные все еще находятся в локальном клонированном Alex репозитории на его локальном устройстве хранения данных. Когда Alex "поместит" данные в репозиторий Git с публичным доступом, измененные данные будут отправлены в этот репозиторий. После того, как на стороне публичного репозитория будет проведена проверка того, могут ли измененные данные быть добавлены в ветвь, в публичном репозитории будут сохранены те же самые объекты, что были ранее созданы в локальном репозитории Git.

На самом деле в сценарии работы репозитория Git присутствует гораздо большее количество взаимодействующих систем, которые находятся на заднем плане и требуют от пользователя подтверждения намерения передачи изменений удаленному репозиторию вне зависимости от локальных механизмов отслеживания изменений. Однако, уровни усложнения архитектуры позволяют команде разработчиков получить в свое распоряжение более гибкую в отношении процессов разработки и публикации систему, как было описано выше в разделе "Начало развития проекта Git".

В сценарии использования системы Subversion участник процесса разработки не должен помнить о необходимости помещения данных в публичный удаленный репозиторий, когда данные готовы для обзора другими участниками. В момент, когда небольшая модификация файла большого размера отправляется в центральный репозиторий Subversion, хранение файлов различий гораздо эффективнее хранения всего содержимого файла каждой версии. Однако, как мы увидим позднее, существует обходной путь, который используется системой Git для решения этой проблемы.

6.4. Тулкит

На сегодняшний день экосистема проекта Git включает множество инструментов с интерфейсом командной строки и другими пользовательскими интерфейсами для работы под управлением множества операционных систем (включая ОС Windows, поддержка которой изначально была неполной). Большинство этих инструментов разработано с использованием основного тулкита Git.

Из-за того, что разработка Git на начальных этапах осуществлялась Linus Torvalds и ввиду связи последнего с сообществом разработчиков Linux, она осуществлялась в соответствии с философией создания тулкитов в большей степени в соответствии с традицией создания инструментов с интерфейсом командной строки для ОС семейства Unix.

Тулкит Git разделен на две части: вспомогательные (*plumbing*) и основные (*porcelain*) функции. Категория вспомогательных функций состоит из низкоуровневых команд, позволяющих осуществлять основные операции, связанные с отслеживанием состояния данных и манипуляциями с направленными ациклическими графами (DAG). Набор основных функций меньше и содержит команды `git`, которые скорее всего понадобятся большинству пользователей системы Git для управления репозиториями и осуществления обмена данными между репозиториями в ходе совместной работы.

Хотя архитектура тулкита и позволила реализовать достаточное количество команд для предоставления доступа к большинству функций разработчикам сценариев, разработчики приложений жаловались на отсутствие пригодной для связывания библиотеки Git. Так как исполняемый файл Git вызывает функцию `die()` код не является реентерантным, поэтому графические и веб-интерфейсы, а также длительно работающие службы должны вызывать функции `fork/exes` для запуска бинарного файла Git, что может замедлить работу приложения.

Для исправления ситуации, с которой столкнулись разработчики приложений, была проведена работа; если вас интересуют подробности, обратитесь к разделу "Что дальше?".

6.5. Репозиторий, данные индексирования и рабочие области

Давайте приступим к непосредственному углубленному исследованию Git на примере локального репозитория для понимания всего лишь нескольких фундаментальных концепций.

Вначале для создания нового инициализированного репозитория Git в нашей локальной файловой системе (при условии использования Unix-подобной операционной системы) мы можем выполнить следующие команды:

```
$ mkdir testgit
$ cd testgit
$ git init
```

Теперь в нашем распоряжении пустой, но инициализированный Git-репозиторий, расположенный в директории `testgit`. Мы можем создавать ветви, добавлять данные, создавать тэги и даже обмениваться данными с другими локальными или удаленными репозиториями Git. Возможен даже обмен данными с репозиториями других типов систем контроля версий при условии использования небольшого набора специальных команд приложения `git`.

Команда `git init` создает поддиректорию `.git` в директории `testgit`. Давайте рассмотрим ее содержимое:

```
tree .git/
.git/
|-- HEAD
|-- config
|-- description
|-- hooks
|   |-- applypatch-msg.sample
|   |-- commit-msg.sample
|   |-- post-commit.sample
|   |-- post-receive.sample
|   |-- post-update.sample
|   |-- pre-applypatch.sample
|   |-- pre-commit.sample
|   |-- pre-rebase.sample
|   |-- prepare-commit-msg.sample
|   |-- update.sample
|-- info
|   |-- exclude
|-- objects
|   |-- info
|   |-- pack
|-- refs
    |-- heads
    |-- tags
```

Находящаяся на верхнем уровне директория `.git` по умолчанию является поддиректорией корневой рабочей директории `testgit`. Она содержит несколько различных типов файлов и директорий:

- Файлы конфигурации (*Configuration*): файлы `.git/config`, `.git/description` и `.git/info/exclude` участвуют в процессе конфигурации локального репозитория.
- Директория сценариев (*Hooks*): директория `.git/hooks` содержит сценарии, которые могут выполняться в моменты наступления определенных событий в рамках жизненного цикла репозитория.
- Файл рабочего пространства (*Staging Area*): файл `.git/index` (которого пока нет в нашем листинге, представленном выше) будет содержать описание рабочего пространства, соответствующего нашей рабочей директории.

- База данных объектов (*Object Database*): директория `.git/objects` является стандартной базой данных объектов системы Git, которая содержит все данные или указатели на локальные данные. Все объекты являются неизменяемыми с момента создания.
- Директория ссылок (*References*): директория `.git/refs` является стандартным местом для хранения указателей, связывающихся и на локальные, и на удаленные ветви, а также ветви с тегами и рабочие ветви. Ссылка является указателем на объект, обычно типа `tag` или `commit`. Управление ссылками осуществляется вне базы данных объектов для возможности изменения ссылок в процессе развития репозитория. В особых случаях ссылки могут указывать на другие ссылки, причем примером такого случая является ветвь `HEAD`.

Директория `.git` на самом деле является репозиторием. Директория, в которой хранится набор рабочих файлов, является рабочей директорией (*working directory*), которая обычно является родительской для директории `.git` (или для *репозитория*). В том случае, если бы вы создавали удаленный репозиторий Git, в котором не было рабочей директории, вам пришлось бы инициализировать его с помощью команды `git init --bare`. Эта команда позволила бы просто создать директорию репозитория непосредственно в корневой директории вместо создания репозитория в форме поддиректории рабочей директории с файлами.

Другим очень важным файлом является файл индексирования Git (*Git Index*): `.git/index`. Он позволяет создать временное рабочее пространство между локальной рабочей директорией и локальным репозиторием. Индексирование подразумевает хранение данных специфичных изменений в одном файле (или в большем количестве файлов), предназначенных для последующего применения. Даже в том случае, если вы вносите изменения, относящиеся к разным типам функций, эти изменения могут быть внесены в исходный код в рамках единственной операции, сопровождающейся подробным пояснением, позволяющим логически разделить их. Для выборочного сохранения определенных изменений в файле или наборе файлов вы можете использовать команду `git add -p`.

Данные изменений Git по умолчанию хранятся в единственном файле, расположенному в директории репозитория. Пути к этим трем пространствам хранения данных в рамках дерева директорий могут задаваться с помощью переменных окружения.

Полезно понимать принципы взаимодействий, осуществляющихся между этими тремя пространствами (репозиторием, пространством индексирования и рабочим пространством) в процессе выполнения нескольких основных команд Git:

- `git checkout [ветвь]`
Эта команда приведет к переводу ссылки `HEAD`-ветви с локального репозитория на текущую ветвь (т.е. `refs/heads/master`), заполнению файла индексирования данными для выбранной ветви и обновлению содержимого рабочей директории для соответствия данным выбранной ветви.
- `git add [файлы]`
Эта команда приведет к созданию перекрестных ссылок с использованием контрольных сумм между *файлами*, описанными в файле индексирования для того, чтобы установить необходимость обновления файла индексирования для измененных файлов в соответствии с версией рабочей директории. В директории Git (или репозитории) ничего не изменится.

Давайте установим более конкретное значение этих утверждений, исследовав содержимое файлов в директории `.git` (или репозитории).

```
$ GIT_DIR=$PWD/.git
$ cat $GIT_DIR/HEAD

ref: refs/heads/master

$ MY_CURRENT_BRANCH=$(cat .git/HEAD | sed 's/ref: //g')
$ cat $GIT_DIR/$MY_CURRENT_BRANCH

cat: .git/refs/heads/master: No such file or directory
```

Мы столкнулись с ошибкой из-за того, что перед добавлением любых данных в репозиторий Git не задается никаких ветвей кроме стандартной для Git ветви с именем `master`, причем самой ветви может и не существовать.

Теперь, когда мы снова попытаемся добавить данные, ветвь `master` будет создана по умолчанию для выполнения данной операции. Давайте выполним операцию (продолжая работу в той же командной оболочке для сохранения истории команд и контекста):

```
$ git commit -m "Initial empty commit" --allow-empty
$ git branch
* master
$ cat $GIT_DIR/$MY_CURRENT_BRANCH
3bce5b130b17b7ce2f98d17b2998e32b1bc29d68
$ git cat-file -p $(cat $GIT_DIR/$MY_CURRENT_BRANCH)
```

Здесь мы наблюдаем представление данных в рамках базы данных объектов Git.

6.6. База данных объектов

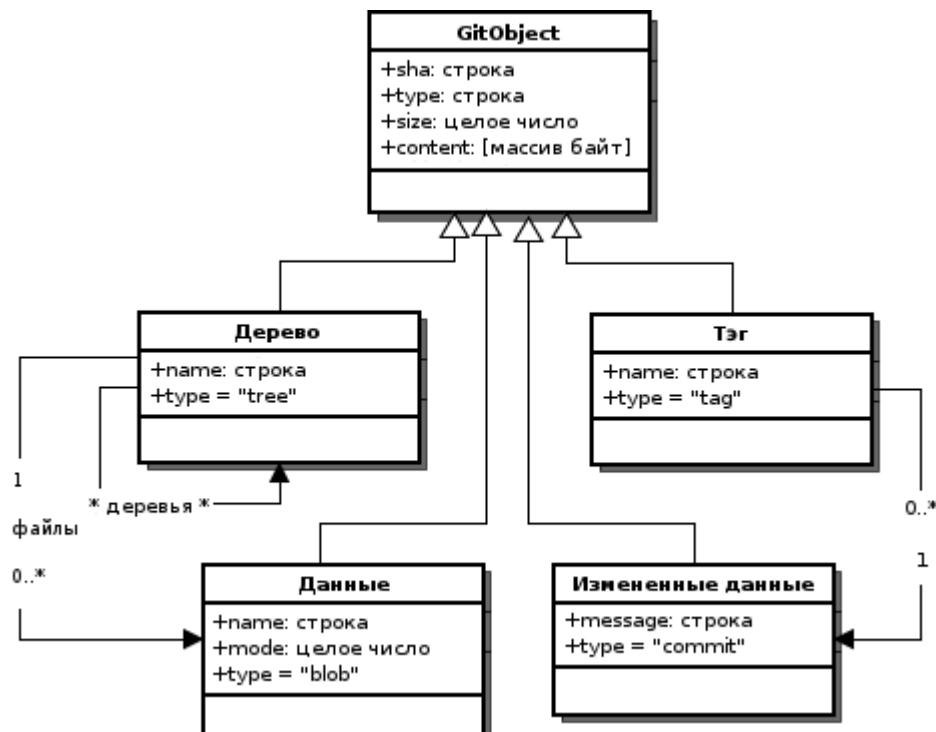


Рисунок 6.2: Объекты Git

В Git используются 4 основных типа примитивных объектов для каждого из типов данных, на основе которых строится локальный репозиторий. Объект каждого типа имеет следующие атрибуты: тип (`type`), размер (`size`) и данные (`content`). Примитивные типы объектов:

- Дерево (*Tree*): элемент дерева может являться данными или еще одним деревом в случае представления директории с данными.
- Данные (*Blob*): данный тип объекта используется для представления файла, находящегося в репозитории.
- Измененные данные (*Commit*): данный тип объекта используется для указания на дерево, представляющее директорию верхнего уровня для измененных в ходе текущей операции данных, а также для ранее измененных данных и стандартных атрибутов.

- Тэг (*Tag*): тэг имеет имя и указывает на операцию изменения данных в истории изменений репозитория, которую он представляет.

Для ссылок на все примитивные объекты используются SHA-хэши, являющиеся идентификаторами объектов из 40 цифр, которые имеют следующие свойства:

- В том случае, если два объекта являются идентичными, они будут иметь одинаковые SHA-хэши.
- В том случае, если объекты различаются, они будут иметь разные SHA-хэши.
- В том случае, если объект был только частично скопирован или произошло другое повреждение данных, повторный расчет SHA-хэша для данного объекта поможет обнаружить такое повреждение данных.

Два первых свойства SHA-хэшей, относящиеся к установлению идентичности объектов, полезны для реализации распределенной модели Git (вторая задача системы Git). Последнее свойство позволяет организовать систему защиты от повреждения данных (третья задача Git).

Несмотря на желаемые результаты использования хранилища на основе направленных ациклических графов для хранения данных и истории объединений ветвей, для многих репозиториев хранилище на основе данных отличий файлов окажется более эффективным в плане использования дискового пространства, нежели хранилище на основе *отдельных* объектов, представленных направленными ациклическими графиками.

6.7. Техники хранения и сжатия данных

Git решает проблему чрезмерного потребления дискового пространства путем сжатия объектов, причем файл индексирования данных используется для указания на сдвиги, по которым находятся определенные объекты в соответствующем упакованном файле.

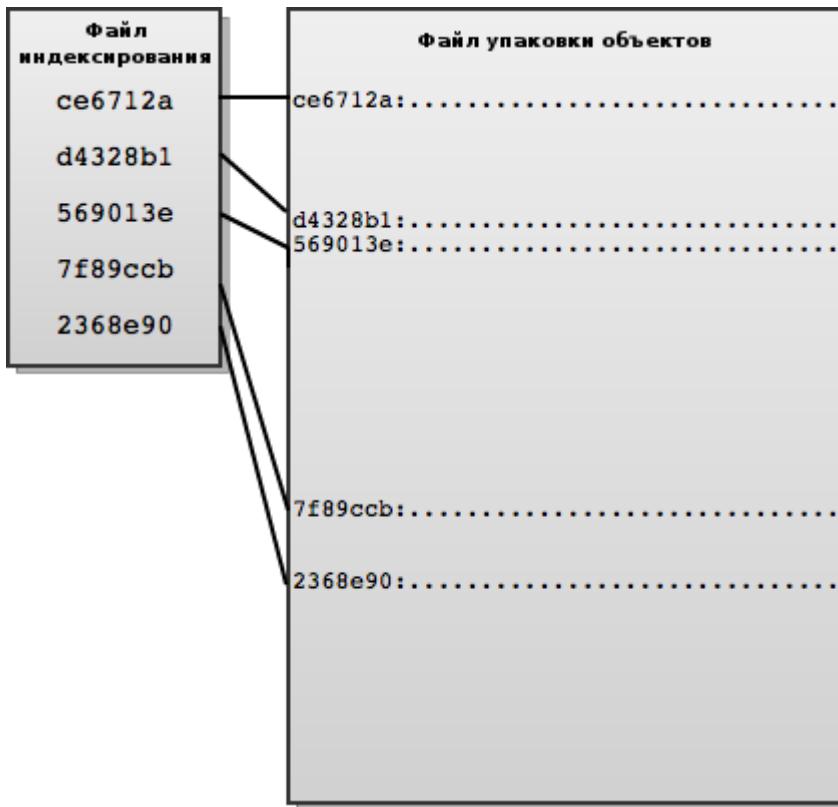


Рисунок 6.3: Диаграмма упакованного файла с соответствующим файлом индексирования

Мы можем подсчитать количество отдельных (или не сжатых) объектов в локальном репозитории Git с помощью команды `git count-objects`. После этого мы можем упаковать отдельные объекты средствами Git в рамках базы данных объектов, удалить уже упакованные отдельные объекты и

обнаружить лишние упакованные файлы с помощью вспомогательных команд Git в случае необходимости.

Формат файла упаковки объектов эволюционировал с начального уровня, представленного форматом с сохранением контрольных сумм для упакованного файла и файла индексирования в самом файле индексирования. Однако, данный подход допускал возможность неустановливаемого повреждения сжатых данных, так как фаза повторной упаковки не предусматривала каких-либо дополнительных проверок. Во 2 версии формата файла упаковки объектов эта проблема была преодолена путем включения контрольных сумм для каждого из сжимаемых объектов в файл индексирования. Версия 2 также позволила создавать упакованные файлы размером более 4 ГБ, которые не поддерживались в начальной версии. Для быстрого установления факта повреждения файла упаковки объектов в его конце должен быть записан 20-байтный хэш SHA-1 для упорядоченного списка всех хэшей SHA объектов из этого файла. В новом формате файла упаковки объектов наибольшее внимание было уделено достижению второй цели проектирования, а именно защите данных от повреждения.

При удаленном взаимодействии Git подсчитывает объем добавленных и содержащихся в репозитории данных, которые должны быть переданы по сети для синхронизации репозиториев (или только ветви) и в процессе работы генерирует файл упаковки объектов, который должен быть отправлен назад с использованием выбранного клиентом протокола.

6.8. История объединения ветвей

Как было сказано ранее, подход системы Git к хранению истории объединения ветвей фундаментально отличается от подхода семейства систем контроля версий на основе RCS. Система Subversion, например, представляет историю изменения файла или дерева директорий в виде линейной последовательности; в этом случае любой элемент файловой системы с большим номером ревизии будет заменять элемент с меньшим номером ревизии. Непосредственное выделение ветвей не поддерживается и может осуществляться только путем самостоятельного создания специальной структуры директорий в репозитории.

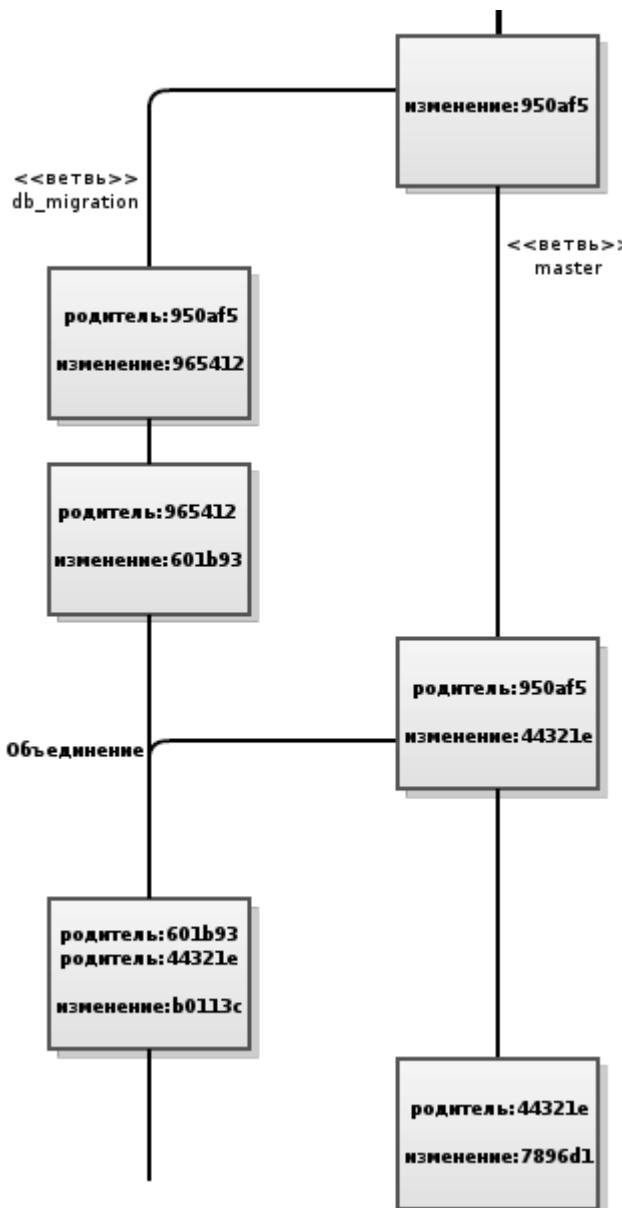


Рисунок 6.4: Диаграмма, показывающая ход процесса объединения ветвей

Давайте используем пример для демонстрации проблем, которые могут возникать при поддержке множества ветвей для одной и той же работы. После этого мы рассмотрим сценарий, показывающий возникающие ограничения.

При работе с "ветвью" в системе Subversion в рамках стандартной корневой директории `branches/branch-name` мы работаем с деревом поддиректорий директории `trunk` (обычно в эквивалентной директории располагается код ветви `master`). Предположим, что в рамках этой ветви производится параллельная разработка программного компонента, изначально разрабатываемого в рамках директории `trunk`.

К примеру, мы можем изменять кодовую базу приложения для поддержки другого типа базы данных. После частичной реализации необходимых функций у нас может возникнуть желание добавить в основную ветвь разработки код из поддиректории другой ветви (не относящийся к коду из директории `trunk`). Мы объединим изменения, прибегнув в случае необходимости к ручной правке, и продолжим нашу разработку. В конце концов мы закончим реализацию нашего кода для миграции на отличный тип базы данных в рамках ветви `branches/branch-name` и объединим код ветви с кодом из директории `trunk`. Проблема, с которой столкнутся пользователи таких систем контроля версий, как Subversion с линейным представлением истории изменений, заключается в том, что не

существует способа получения информации о том, какие какие изменения из других ветвей были добавлены в код из директории `trunk` ранее.

Системы контроля версий, записывающие историю объединений ветвей с помощью направленных ациклических графов, такие, как Git, обрабатывают подобные сценарии достаточно хорошо. Предполагая, что другая ветвь не содержит изменений, которые были внесены в нашу ветвь с реализацией кода миграции на другой тип базы данных (скажем, ветви `db-migration` из нашего репозитория Git), на основе взаимосвязей родительских объектов мы можем установить что изменения, внесенные в ветвь `db-migration` содержали ссылку (`tip` или `HEAD`) на другую ветвь, в которой ведется основная разработка. Следует отметить, что объект добавления данных может не иметь или иметь произвольное количество (ограниченное только возможностями объединяющего ветки пользователя) родительских объектов. Следовательно, в случае объединения ветвей ветвь `db-migration` получит информацию о том, будет ли она объединяться с текущей ветвью или с ветвью, в рамках которой производится основная разработка, получив хэши SHA родительских объектов. Это же утверждение актуально и для объединения с ветвью `master` (эквивалентом директории `trunk` при работе с системой Git).

Вопрос, на который сложно ответить при использовании истории объединений ветвей на основе направленных ациклических графов (и линейных представлений) заключается в том, какие изменения содержатся в каждой из ветвей. Например, в сценарии выше мы предположили, что все изменения из обоих рассматриваемых ветвей были внесены в каждую из ветвей. Но это не всегда так.

При работе в более простых ситуациях Git может извлекать изменения из других ветвей и вносить их в текущую ветвь, но только в том случае, если изменение может быть непосредственно применено без дополнительного редактирования.

6.9. Что дальше?

Как говорилось ранее, основная часть системы Git в сегодняшнем виде создана в соответствии с философией архитектуры тулкита из мира Unix, которая очень хорошо подходит для сценариев, но менее полезна в случае интеграции или связывания с приложениями или службами, работающими в течение длительного периода времени. Хотя поддержка Git на сегодняшний день реализована в множестве интегрированных сред разработки, работа по добавлению кода поддержки и его сопровождению оказывается более сложной, чем работа, связанная с интеграцией кода поддержки других систем контроля версий, которые предоставляют библиотеки для множества платформ, которые упрощают связывание кода и разделение функций.

Для решения этой проблемы Shawn Pearce (из подразделения Google Open Source Programs Office) возглавил проект по созданию библиотеки Git, которая могла связываться с кодом приложений и распространялась под более либеральной лицензией, не препятствующей ее использованию. Эта библиотека получила имя [libgit2](#). Она не получила заметного распространения до того момента, как студент по имени Vincent Marti не выбрал ее для работы в рамках проекта Google Summer of Code в прошлом году. С того момента Vincent Marti и инженеры компании Github продолжили работу над проектом libgit2 и создали биндинги для множества других популярных языков программирования, таких, как Ruby, Python, PHP, языки .NET, Lua и Objective-C.

Shawn Pearce также начал разработку библиотеки с именем [JGit](#) на языке Java, которая распространялась под лицензией BSD и поддерживала большое количество стандартных операций, выполняемых при работе с репозиториями Git. На данный момент развитие этой библиотеки поддерживается организацией Eclipse Foundation и она используется для работы среды интегрированной разработки Eclipse с системой Git.

Другие интересные и экспериментальные проекты с открытым исходным кодом, разрабатываемые вне основного проекта Git представлены множеством реализаций, использующих альтернативные хранилища данных в качестве систем организации базы данных объектов Git, среди которых можно выделить:

- [jgit_cassandra](#), позволяющий на постоянной основе хранить объекты Git в гибридном хранилище данных под названием Apache Cassandra, используя распределение данных в стиле Dynamo совместно с семантиками модели данных семейств столбцов BigTable.
- [jgit_hbase](#), позволяющий осуществлять операции чтения и записи данных объектов Git, хранимых в распределенном хранилище данных HBase, используя пары ключ-значение.
- [libgit2-backends](#), продолжающий начатую в рамках проекта libgit2 работу по созданию хранилищ данных объектов Git на основе множества популярных систем хранения данных, таких, как Memcached, Redis, SQLite и MySQL.

Все эти проекты с открытым исходным кодом развиваются независимо от основного проекта Git.

Как вы видите, на сегодняшний день существует большое количество методов использования формата хранения данных, применяемого в Git. Лицом проекта Git сейчас является не только интерфейс командной строки тулкита, развивающегося в рамках основного проекта Git; это также формат репозитория и протокол, используемый для осуществления обмена данными между репозиториями.

В момент написания этой главы большинство названных проектов, в соответствии с заявлениями их разработчиков, не достигло стабильных релизов, поэтому все еще требуется выполнение некоторого объема работы в этой области, но будущее проекта Git все равно кажется многообещающим.

6.10. Выученные уроки

При разработке программного обеспечения любое архитектурное решение по своей сути является компромиссом. Наблюдая с позиции опытного пользователя системы контроля версий Git, а также того, кто разрабатывает программное обеспечение для работы с моделью базы данных объектов Git, я испытываю гордость по поводу сегодняшнего состояния системы Git. Система стала такой благодаря урокам, извлеченным при исправлении вызывающих наибольшее количество повторяющихся жалоб недоработок, возникающих в результате принятия архитектурных решений разработчиками основной части проекта Git.

Одной из наиболее часто возникающих жалоб от разработчиков и руководителей проектов, которые используют Git, является жалоба на отсутствие интеграции с интегрированными средами разработки, реализованной так же хорошо, как и в случае других инструментов контроля версий. Архитектура тулкита, выбранная при реализации Git, сделала эту работу более сложной, чем работа по интеграции других современных систем контроля версий с интегрированными средами разработки и связанными с ними инструментами.

Ранее некоторые команды Git были реализованы в форме сценариев оболочки. Эти реализации команд в форме сценариев сделали систему Git менее переносимой, особенно на платформу Windows. Я уверен, что разработчики основной части проекта Git не упустили из вида этот факт, но он в конечном счете негативно повлиял на внедрение Git в крупных организациях ввиду распространенных сложностей, возникавших при переносе системы на другие платформы на ранних стадиях разработки Git. Сегодня существует проект "Git for Windows", реализуемый добровольцами с целью своевременного переноса новых версий Git на платформу Windows.

Косвенным последствием проектирования Git на основе архитектуры тулкита с большим набором вспомогательных команд является неуверенность новых пользователей, которая возникает почти сразу же после начала использования системы; эта неуверенность возникает ввиду множества факторов от растерянности при виде всех доступных вспомогательных команд до непонимания

сообщений об ошибках, выводимых после некорректного завершения низкоуровневой вспомогательной задачи, при этом существует еще очень много областей, в которых новые пользователи могут сбиться с пути. Это обстоятельство осложнило внедрение Git некоторыми командами разработчиков.

Даже при наличии всех этих жалоб на систему Git, я очень вдохновлена возможностями, которые откроются в ходе процесса разработки проекта Git в будущем, а также в ходе разработки всех связанных проектов с открытым исходным кодом, которые были начаты благодаря появлению Git.

7.Проект GPSD

Глава 7 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Проект GPSD представляет собой набор инструментальных средств для управления коллекциями устройств GPS и другими датчиками, связанными с навигацией и хранением точного времени, в том числе морских радиосистем автоматической идентификации AIS (Automatic Identification System) и цифровых компасов. Основная программа — демон сервиса, называемый `gpsd`, управляет набором датчиков и формирует от них всех сообщения в виде потока объектов JSON на известном порту TCP/IP. Среди других программ, имеющихся в наборе, есть примеры клиентских программ, используемых в качестве модели кода, а также различные диагностические инструментальные средства.

Проект GPSD развернут на достаточно большом количестве ноутбуков, смартфонов и автономных транспортных средств, в том числе на самоходных автомобилях и подводных лодках - роботах. Его возможности во встраиваемых системах используются для навигации, ведения точного землемерия, позиционно-чувствительной научной телеметрии и сервисов времени, имеющихся в сетях. Он даже используется в идентификационной системе «свой-чужой» боевых бронированных машин, в том числе в основном боевом танке M1 «Абрамс».

Проект GPSD является проектом среднего размера - около 43 тыс. строк кода, написанных, в основном, на C и Python, с историей нынешней лидирующей позиции начиная с 2005 года и с предысторией начиная с 1997 года. Основная стабильная группа его разработчиков состояла из трех разработчиков, с полу-регулярными вкладами приблизительно от более чем двух десятков разработчиков и обычными одноразовыми патчами от сотни других.

Исторически сложилось так, что в проекте GPSD присутствовало исключительно низкое количество ошибок, причем как измеряемое с помощью средств аудита, например, `splint`, `valgrind` и `Coverity`, так и оцениваемое согласно сообщениям об ошибках в собственном трекере, а также в других местах. Это не случайно; в проект очень агрессивно внедрялись технологии автоматизированного тестирования, и эти усилия окупились сторицей.

Проект GPSD достаточно хорошо делает все то, что должен делать, он вобрал в себя или эффективно уничтожил всех своих предшественников и предотвратил, по меньшей мере, одну прямую попытку конкурировать с ним. В 2010 году проект GPSD выиграл первый грант Good Code Grant альянса Alliance for Code Excellence. К тому моменту, как вы закончите читать эту главу, вы должны понять, благодаря чему.

7.1. Почему существует проект GPSD

GPSD существует из-за того, что прикладные протоколы, поставляемые с датчиками GPS и другие датчики, относящиеся к навигации, плохо спроектированы, недостаточно документированы и сильно зависят от типа и модели датчика. Подробное обсуждение смотрите по ссылке [Ray]; там, в

частности, вы узнаете о капризах стандарта NMEA 0183 (варианта стандарта пакетов сообщений GPS) и о бессистемной куче плохо документированных протоколов поставщиков, которые с ним конкурируют.

Если бы приложения справлялись со всей этой сложностью самостоятельно, то результатом было бы огромное количество нестабильного и дублирующего кода, что ведет к высокой степени наличия видимых пользователю дефектов и постоянным проблемам, т. к. аппаратура постепенно становилась бы несовместимой с приложениями.

Проект GPSD изолирует приложения, обрабатывающие данные о местоположении, от деталей аппаратного интерфейса благодаря тому, что в самом проекте известно обо всех протоколах (во время написания статьи мы поддерживали около 20 различных протоколов), и благодаря такому управлению последовательными устройствами и устройствами USB, что приложениям этого делать уже не требуется, а достаточно только получать информацию от датчиков в простом и независимом от устройств формате JSON. GPSD еще больше упрощает жизнь благодаря тому, что предоставляются клиентские библиотеки и клиентским приложениям даже не нужно знать об этом формате сообщений. Вместо того, чтобы получать информацию с датчиков, достаточно просто вызвать процедуру.

Система GPSD также поддерживает хранение точного времени; она может выступать в качестве источника времени для `ntpd` (демона протокола сетевого сервиса времени), если в каком-нибудь из подключенных к нему датчиков имеется возможность выдавать сигналы PPS (секундные сигналы). Разработчики проекта GPSD тесно сотрудничают с проектом `ntpd` с целью улучшения сетевого сервиса времени.

В настоящее время (середина 2011 года) мы работаем над завершением поддержки сети AIS морских навигационных приемников. В будущем мы ожидаем, что будем поддерживать новые виды датчиков, работающие с информацией о местоположении, такие как приемники авиационных систем «запрос-ответ» второго поколения, поскольку станут доступными документация по протоколу и тестовые экземпляры устройств.

Таким образом, самым важным аспектом в проекте GPSD является скрытие всех аппаратно-зависимых безобразий за простым клиентским интерфейсом, не требующим для получения данных никаких настроек.

7.2. Взгляд извне

Основной программой в наборе инструментальных средств GPSD является демон сервиса `gpsd`. Он может собирать данные, поступающие от набора подключенных датчиков устройств через соединения RS232, USB, Bluetooth, TCP/IP и UDP. Данные обычно поступают на порт 2947 на TCP/IP, но также могут поступать через совместно используемую память или интерфейс D-BUS.

Пакет GPSD поставляется с клиентскими библиотеками для языков C, C++ и Python. Он включает в себя образцы клиентских приложений на языках C, C++, Python и PHP. Клиентская сборка для Perl доступна через CPAN. Эти клиентские библиотеки не только удобны для разработчиков приложений; они также спасают разработчиков GPSD от головной боли, изолируя приложения от деталей сообщений JSON пакета GPSD. Таким образом, интерфейс API, предоставляемый клиентам, может оставаться такими же даже в случае, когда в протокол для новых типов датчиков добавляются новые возможности.

К числу других программ в наборе инструментальных средств относятся утилита низкоуровневого мониторинга устройств (`gpsmon`), профилировщик, который создает отчеты о статистике ошибок и синхронизации устройств (`gpsprof`), утилита настройки устройств (`gpsctl`), а также программа для потокового преобразования журнальных данных в удобочитаемый формат JSON (`gpsdecode`).

Вместе они помогают технически подкованным пользователями заглянуть как можно глубже в работу присоединенных датчиков, работе которых у них вызывает беспокойство.

Конечно, эти инструментальные средства также помогают самим разработчикам проекта GPSD проверять правильность работы `gpsd`. Одним из самых важных тестовых инструментальных средств является `gpsfake`, средство тестирования `gpsd`, которое можно подключить к журнальным файлам любого количества датчиков так, как будто бы они являются реальными устройствами. С помощью `gpsfake` мы можем повторно запустить журналный файл датчика, полученный вместе с сообщением об отказе, и воспроизводить конкретные проблемы. `gpsfake` также является движком нашего обширного инструментального средства, предназначенного для регрессионных тестов, которое снижает затраты на модификацию программного обеспечения благодаря более простому обнаружению изменений, из-за которых возникает проблема.

Одним из самых важных уроков, который, как мы думаем, мы получили для будущих проектов, состоит в том, что недостаточно, чтобы набор инструментальных средств был корректным, должна также быть возможность *продемонстрировать его правильность*. Мы обнаружили, что, когда эта цель реализуется должным образом, она будет не маскировочной сеткой, а парой крыльев — время, которое мы затрачиваем на написание нагрузочных и регрессионных тестов, будет много-кратно окупаться сторицей благодаря той свободе, которая позволяет нам изменять код, не опасаясь, что мы создадим трудно обнаруживаемый беспорядок в существующих функциональных возможностях.

7.3. Слои программного обеспечения

Внутри GPSD происходит гораздо больше того, чем могут предположить люди, знающие на практике, что нужно «подключить датчик и он просто заработает». Внутренняя структура `gpsd` естественным образом делится на четыре части: *драйверы, анализатор пакетов, основную библиотеку и мультиплексор*. Мы опишем эти части, рассматривая их снизу вверх.

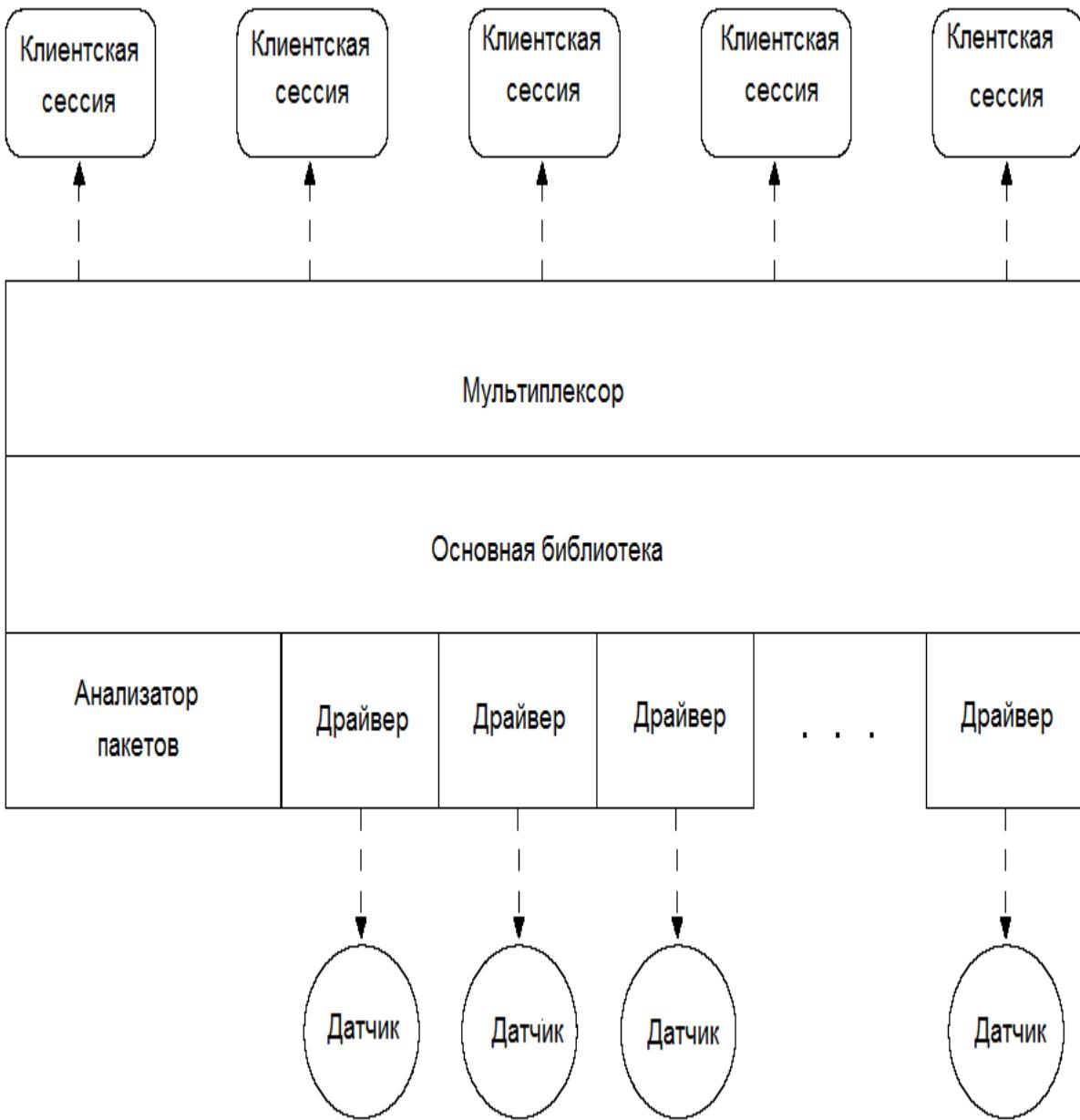


Рис.7.1: Слои программного обеспечения

Драйверы являются, в сущности, драйверами устройств пользовательского пространства для каждого вида чипсетов датчиков, которые мы поддерживаем. Основным их ключевыми элементами являются методы анализа пакетов данных и получения информации о времени, позиции скорости или состоянии датчиков, изменения их режимов работы или скорости передачи данных, определения подтипов устройств и т.д. Вспомогательные методы могут поддерживать операции управления драйверами, например, изменение скорости последовательного обмена данными с устройством. Весь интерфейс драйвера является структурой на языке C, заполненной указателями на данные и методы, которая преднамеренно имитирует структуру драйверов устройств в Unix.

Анализатор пакетов отвечает за получение пакетов данных из последовательных входных потоков. Это, большей частью, машина состояний или конечный автомат, который наблюдает за всем, что выглядит как один из наших 20 или около того известных типов пакетов (в большинстве из них есть контрольная сумма, так что когда мы говорим, что обнаружили один из них, мы говорим это с высокой уверенностью). Поскольку устройства могут подключаться в режиме горячей замены или изменять режимы, тип пакета, поступающего из последовательного порта или порта USB, не обязательно постоянно будет совпадать с типом первого пакета.

Основная библиотека (core library) осуществляет управление сессией работы с устройством-датчиком. Основными ключевыми аспектами являются следующие:

- открытие сессии с помощью открытия устройства и чтения данных из него, перебор комбинаций скорости передачи и значений четности/стопового бита до тех пор, пока анализатор пакетов не будет синхронизирован с известным ему типом пакета;
- опрос устройства, получающего пакеты, и
- закрытие устройства и завершение сессии.

Ключевой особенностью основной библиотеки является то, что она отвечает за то, чтобы для каждого подключения GPS использовался правильный драйвер устройства, зависящий от типа пакетов, возвращаемого анализатором пакетов. Это *не настраивается заранее* и может изменяться с течением времени, особенно если в устройстве есть возможность переключения между различными протоколами (в большинство чипсетов GPS поддерживается протокол NMEA и один или несколько двоичных протоколов, предоставляемых поставщиком, а такие устройства, как AIS, могут по одному и тому же проводу пересыпать пакеты, использующие два различных протокола).

Наконец, *мультиплексор* является частью демона, который работает с сессиями клиентов и назначением устройств. Он отвечает за передачу сообщений клиентам, принимая команды клиента и реагируя на сообщения о горячих подключениях. В сущности, все это задается в одном исходном файле `gpsd.c` и никакое из заданий не обращается к драйверам устройств напрямую.

Первые три компонента (кроме мультиплексора) компонуются друг с другом в библиотеке с названием `libgpsd` и ими можно пользоваться отдельно от мультиплексора. Наши прочие инструментальные средства, с помощью которых происходит непосредственное общение с датчиками, например, `gpsmon` и `gpsctl`, осуществляют это с помощью непосредственного обращения к основной библиотеке и слою драйверов.

Наиболее сложным отдельным компонентом является анализатор пакетов, имеющий размер около двух тысяч строк кода. Их количество сократить нельзя; конечный автомат, который может распознавать такое количество различных протоколов, как это он делает, обязательно будет большим и запутанным. К счастью, анализатор пакетов также легко изолировать и протестировать; проблемы, имеющиеся в нем, не будут влиять на другие части кода.

Слой мультиплексора имеет приблизительно такой же размер, но он немного менее запутанный. Основную часть демона составляют драйверы устройств - около 15 тыс. строк. Оставшаяся часть кода - все средства и библиотеки поддержки вместе с инструментальными средствами тестирования клиентов - вместе по размеру равны примерно размеру демона (некоторый код, в частности парсер JSON, используется совместно как в демоне, так и в клиентских библиотеках).

Преимущество такого подхода, использующего слои, демонстрируется несколькими способами. Во-первых, поскольку драйверы для новых устройств писать легко, некоторые из них были отданы на разработку лицам, не входящим в основную команду разработчикам: интерфейс API драйвера документирован, а отдельные драйверы подключаются к основной библиотеке только через указатели, расположенные в главной таблице типов устройств.

Еще одним преимуществом является то, что системные интеграторы могут резко сократить размер GPSD при использовании во встроенных системах, просто решив не компилировать неиспользуемые драйверы. Демон для начала не так уж и велик и сборка, урезанная соответствующим образом, работает вполне нормально на маломощных низкоскоростных устройствах ARM, не обладающих большим количеством памяти. ARM является RISC архитектурой с 32-битным набором команд, используемой в мобильных устройствах и во встраиваемой электронике. Смотрите http://en.wikipedia.org/wiki/ARM_architecture.

Третье преимущество слоев состоит в том, что мультиплексор демона может быть отсоединен от основной библиотеки и заменен простой логикой, например, такой, которая в потоке преобразует сообщения, поступающие в журнальный файл, в сообщения JSON, т.е. именно то, что делает утилита `gpsdecode`.

В этой части архитектуры GPSD нет ничего нового. Ее урок состоит в том, что сознательное и строгое применение шаблона проектирования, предназначенного для устройств Unix, приносит пользу не только в ядре ОС, но и в пользовательских программах, в которых они также необходимы при работе с разнообразными аппаратными средствами и протоколами.

7.4. Поток данных

Теперь мы рассмотрим архитектуру GPSD с точки зрения потока данных. В режиме нормальной работы `gpsd` представляет собой цикл, ожидающий входных данных от одного из следующих источников:

1. Набора клиентов, передающих запросы через порт TCP/IP.
2. Набора навигационных датчиков, подключенных через последовательные устройства или устройства USB.
3. Специального управляющего сокета, используемого скриптами горячего подключения и некоторыми конфигурационными инструментальными средствами.
4. Некоторого количества серверов, периодически выдающих корректирующие сообщения для GPS (DGPS и NTRIP). Эти сообщения будут обрабатываться точно также, как если бы они поступали от навигационных датчиков.

Когда порт USB активируется устройством, которое может быть навигационным датчиком, скрипт горячего подключения (поставляется с GPSD) отправляет уведомление на управляющий сокет. Это сигнал для слоя мультиплексора поместить устройство в свой внутренний список датчиков. И, наоборот, событие, связанное с удалением устройства, может удалять устройство из этого списка.

Когда клиент выдает запрос о том, что наступило время, слой мультиплексора открывает навигационные датчики в своем списке и начинает принимать от них данные (добавляя дескрипторы их файлов в набор в вызове `main`). В остальных случаях все устройства GPS закрыты (но остаются в списке) и демон находится в покое. Для устройств, которые прекратили отправку данных, в списке устройств устанавливается таймаут.



Рис.7.2: Поток данных

Когда данные поступают от навигационного датчика, они подаются в анализатор пакетов - конечный автомат, который работает как лексический анализатор компилятора. Работа анализатор пакетов состоит в накапливании данных из каждого порта (по отдельности) и определении, когда накопленные данные станут представлять собой пакет известного типа.

В пакете могут находиться данные GPS, указывающие местоположение, датаграммы marine AIS, данные с датчика магнитного компаса, широковещательные пакеты DGPS (дифференциальная GPS) или некоторые другие данные. Анализатор пакетов не заботится о содержании пакета; все,

что он делает, это сообщает основной библиотеке, что он накопил данные и передает пакет и тип пакета.

Затем основная библиотека перенаправляет пакет в драйвер, связанный с этим типом пакетов. Работа драйвера состоит в получении данных из пакета и помещения их в структуру, созданную в рамках сессии для конкретного устройства, а также установке некоторых битов состояния, сообщающих слою мультиплексора о том, какие данные он получил.

Один из этих битов указывает на то, что демон собрал достаточно данных для того, чтобы отправить ответ своим клиентам. Когда этот бит устанавливается после считывания данных с датчика устройства, то это означает, что мы достигли конца пакета, конца группы пакетов (которая может состоять из одного или нескольких пакетов), и данные, находящиеся в сессионной структуре устройства, должны быть переданы в один из механизмов экспорта.

Основным экспортным механизмом является «сокет»; он создает объект сообщения в формате JSON и отправляет его всем клиентам, наблюдающим за устройством. Есть еще экспортный механизм с совместно используемой памятью, куда копируются данные вместо того, чтобы копировать их в совместно используемую память сегмента. В любом из этих случаев, предполагается, что клиентская библиотека распакует данные в структуру в памяти клиентской программы. Также доступен третий экспортный механизм, который сообщает о новом местоположении через шину DBUS.

Код GPSD распределен по горизонтали так же тщательно, как и по вертикали. Анализатор пакетов не знает и не должен знать ничего о том, чем нагружен пакет, и не должен беспокоиться о том, является ли его источником порт USB, устройство RS232, радиоканал Bluetooth, псевдо-терминал tty, соединение с сокетом TCP или поток пакетов UDP. В драйверах известно, чем загружен пакет, но ничего не известно о внутренних особенностях анализа пакетов и о экспортных механизмах. Экспортные механизмы рассматриваются только как сессионные структуры данных, обновляемые драйверами.

Такое разделение функций очень хорошо служит проекту GPSD. Например, когда в начале 2010 года мы получили запрос адаптировать код так, чтобы он для бортовой навигационной системы робота - подводной лодки принимал данные от датчика, поступающие в виде пакетов UDP, это удалось легко сделать с помощью нескольких строк кода без нарушения последующих этапов конвейера обработки данных.

В более общем смысле тщательное разделение на слои и модули позволило относительно легко добавлять новые типы датчиков. Мы добавляем новые драйверы приблизительно каждые шесть месяцев; причем некоторые из них были написаны людьми, которые не входят в состав основных разработчиков.

7.5. Защищая архитектуру

По мере того как развиваются такие программы с открытым исходным кодом, как gpsd, одной из повторяющихся проблем становится проблема, связанная с тем, что каждый участник для того, чтобы решить его конкретную проблему, будет делать то, что вызовет постепенное перемещение все большего количества информации между слоями или стадиями, между которыми первоначально были установлены строгие границы.

На момент написания статьи мы обеспокоены тем, что, возможно, некоторую информацию о типе источника входных данных (USB, RS232, pty, Bluetooth, TCP, UDP) потребуется передавать в слой мультиплексора для того, чтобы ему сообщить, должны ли, например, в неопознанное устройство посыпаться строки, позволяющие опознать устройство. Такие строки иногда необходимы для того, чтобы «разбудить» датчики RS232C, однако есть веские причины не отправлять их на любые дру-

гие устройства, где они не нужны. Многие устройства GPS и другие датчики создавались с малым бюджетом и в спешке; некоторые из них можно ввести в состояние ступора, если посыпать им неожиданные управляющие строки.

По аналогичной причине в демоне есть параметр `-b`, который запрещает демону изменять скорость обмена данными во время цикла анализа, выполняемого анализатором пакетов. Некоторые плохо сделанные устройства Bluetooth справляются с такими ситуациями настолько плохо, что для того, чтобы их вновь заставить работать, их надо перезапустить при помощи отключения питания; в экстремальном случае пользователю для того, чтобы сбросить блокировку датчику, потребовалось в действительности отпаять батарейку резервного питания!

Оба этих случая являются необходимыми исключениями из общих правил разработки проекта. Однако, гораздо более обычны случаи, когда такие исключения ведут к плохим ситуациям. Например, у нас были некоторые патчи, предназначенные для того, чтобы служба времени PPS работала лучше, которые нарушали деление на вертикальные слои, что сделало невозможным PPS правильно работать с более чем с одним драйвером, для помощи которым патчи и были предназначены. Мы отказались от них и использовали более сложные устройства, которым не требовались улучшения.

Однажды несколько лет назад нас попросили о поддержке устройства GPS со странной особенностью, состоящей в том, что контрольные суммы в их пакетах NMEA могут быть неправильными в тех случаях, когда устройство не смогло определить местоположение. Для поддержки этого устройства, мы должны были бы либо (а) отказаться от проверки контрольных сумм для любых входящих данных, которые были похожи на пакеты NMEA, рискуя тем, что анализатор пакетов мог передавать мусор в драйвер NMEA, либо (б) добавить параметр командной строки, в котором принудительно указывается тип датчика.

Руководитель проекта (автор этой главы) отказался и от первого и от второго. Очевидно, что отказ от проверки пакетов NMEA был плохим решением. Но переключатель, в котором принудительно указывается тип датчика, было бы приглашением с леностью относиться к правильному автоматическому конфигурированию, что вызвало бы проблемы на всем пути вплоть до клиентских приложений GPSD и их пользователей. Следующим шагом по этому пути, вымощенному благими намерениями, наверняка был бы переключатель скорости обмена данными. Вместо этого, мы отказались поддерживать такое неисправное устройство.

Одной из самых важных обязанностей ведущего архитектора проекта является защита архитектуры от разумных «исправлений», который разрушают проект и будут причиной проблем его работы или сильной головной боли при его обслуживании в будущем. Аргументы, касающиеся этого, могут быть довольно горячими, особенно когда конфликт, связанный с защитой архитектуры, касается какой-либо функции, которая, по мнению разработчика или пользователя, считается незаменимой. Но эти аргументы необходимы, поскольку самый простой вариант часто оказывается неправильным в более долгосрочной перспективе.

7.6. Нет конфигурирования — нет суэты

Чрезвычайно важной особенностью демона `gpsd` является то, что в нем отсутствуют средства конфигурирования (с одним небольшим исключением для устройств Bluetooth с испорченной прошивкой). Нет настроечного файла! Демон определяет типы датчиков, с которыми он общается, путем прослушивания входных данных. Для устройств RS232 и USB `gpsd` даже автоматически определяет скорость обмена данными (то есть, автоматически определяет скорость линии последовательного доступа), так что демону нет необходимости заранее знать скорость/четность/стоповые биты, с которыми датчик подает информацию.

Если в операционной системе, в которой установлен демон, есть возможность горячего подключения, то скрипты горячего подключения могут отправлять сообщения об активации и деактивации устройств в управляющий сокет, который уведомит демон об изменении в его окружении. В дистрибутиве GPSD такие скрипты поставляются для Linux. Результатом является то, что конечные пользователи могут подключить USB-устройство GPS к своему ноутбуку и ожидать, что оно немедленно начать передавать сообщения, которые смогут прочитать приложения, определяющие местоположение - без путаницы, без суеты и без редактирования настроечного файла или реестра свойств.

Преимущества такого подхода распространяется вплоть до стека приложений. Кроме всего прочего, это значит, что пока все это работает, в приложениях, определяющих местоположение, может не быть конфигурационной панели, связанной с настройками GPS и порта. Это экономит много усилий у тех, кто пишет приложения, а также у пользователей: они могут рассматривать определение местоположения как сервис, который почти настолько прост, как системные часы.

Одно из следствий философии отсутствия конфигурационных средств состоит в том, что мы не принимаем предложения о добавлении конфигурационного файла или дополнительных параметров командной строки. Проблема в том, что конфигурационный файл, который можно редактировать, *нужно будет* редактировать. Это значит, что для конечных пользователей добавляются хлопоты, связанные с настройкой, т. е. с тем, чего следует избегать в демоне хорошо спроектированного сервиса.

Разработчики GPSD являются специалистами системы Unix, работающими согласно глубоко укоренившейся традиции Unix, согласно которой конфигурационные средства и наличие большого количества регуляторов близко к религии. Тем не менее, мы думаем, что в проектах с открытым исходным кодом можно попытаться сделать намного больше, выбросив настроечные файлы и автоматически выполняя конфигурирование в зависимости от того, в какой среде происходит действие.

7.7. Ограничения, имеющиеся во встроенных системах, полезны

С 2005 года основной задачей проекта GPSD были разработки встраиваемых систем. Первоначально это было связано с тем, что в нас были сильно заинтересованы системные интеграторы, работающие с одноплатными компьютерами, но затем это проявилось неожиданным образом: установками на смартфонах с поддержкой GPS. Однако, согласно отчетам нашими самыми любимыми встроенными системами все еще остаются роботы-подводные лодки.

Проектирование встраиваемых решений повлияло на GPSD в важных направлениях. Для того, чтобы код хорошо работал в низкоскоростных системах с небольшим количеством памяти с низким энергопотреблением, мы много размышляем об экономии используемой памяти и не сильной загрузке процессора.

Одним из важных аспектов в этом вопросе является, как уже упоминалось, возможность обеспечивать, чтобы в сборках `gpsd` не было ничего лишнего, кроме определенного набора протоколов для датчиков, которые должен поддерживать системный интегратор. В июне 2011 года для минимальной статической сборки `gpsd` для системы x86 необходим был объем памяти равный приблизительно 69К (это со прикомпонованными необходимыми стандартными библиотеками на C) на 64-разрядной архитектуре x86. Для сравнения, статическая сборка со всеми драйверами равна приблизительно 418К.

Другая особенность состоит в том, что мы профилируем хотспоты CPU несколько иначе, чем в большинстве проектов. Поскольку датчики месторасположения, как правило, передают лишь не-

большие объемы данных с интервалом порядка 1 секунды, производительность в обычном смысле не является проблемой для проекта GPSD - даже крайне неэффективный код вряд ли будет настолько увеличивать задержки, что они будут видны на уровне приложений. Вместо этого, наши усилия направлены на уменьшение использования ресурсов процессора и сокращение энергопотребления. В этом вопросе мы оказались довольно успешными: даже на маломощных системах ARM без FPU, доля ресурсов процессора, потребляемая `gpsd`, была снижена приблизительно до уровня шума, вносимого профилировщиком.

Хотя разработка кода, мало занимающего память и с хорошей энергоэффективностью, является в данный момент в значительной степени решенной проблемой, есть еще один аспект, стремление к которому во встроенных системах все еще создает напряженность в архитектуре GPSD: использование скриптовых языков. С одной стороны, мы хотим минимизировать дефекты, связанные с низким уровнем управления ресурсами, удаляя из языка C столько кода, сколько это возможно. С другой стороны, язык Python (предпочитаемый нам скриптовый язык) для большинства встроенных систем просто слишком тяжеловесен и медленителен.

Мы решаем этот компромисс очевидным образом: демон сервиса `gpsd` написан на языке C, в то время как фреймворк тестирования и несколько утилит поддержки написаны на языке Python. Со временем, мы надеемся перенести большую часть вспомогательного кода из C в Python, но из-за встроенных систем такие решения становятся источником постоянных противоречий и неудобств.

Тем не менее, в целом мы считаем, что влияние встроенных систем делает код достаточно прочным. Оно хорошо чувствуется в том, что пишется рациональный компактный код, который экономно использует ресурсы процессора. Говорят, что искусство создается благодаря творчеству в условиях ограничений; под таким влиянием GPSD действительно становится лучшим.

Это чувство не следует превращать непосредственно в совет для других проектов, нужно делать нечто другое: не думать, а измерять! Нет ничего другого, похожего на обычное профилирование и измерение размера занимаемой памяти, предупреждающих вас, когда вы уноситесь мыслью в создании разных наворотов, и убеждающих вас, что что-то не так.

7.8. JSON и архитектонауты

Одно из самых значительных преобразований в истории проекта произошло, когда мы перешли от первоначально используемого протокола сообщений к использованию JSON в качестве метапротокола и передачи клиентам объектов JSON. В исходном протоколе использовались однобуквенные символы в качестве команд и ответов, и т. к. возможности демона постепенно увеличивались, мы буквально задыхались в пространстве односимвольных имен.

Переход на JSON был большой победой. JSON сочетает в себе традиционные достоинства Unix использования чисто текстового формата - легко просматривать, легко редактировать с помощью стандартных инструментов, легко создавать с помощью программ - с возможностью передавать структурированную информацию более богатыми и гибкими способами.

Отобразив типы сообщений в объекты JSON, мы обеспечили, что любое сообщение может содержать объединение строковых, числовых и логические данных и структур (возможность, которой не было в старом протоколе). Благодаря тому, что типы сообщений идентифицируются с помощью атрибута `class`, мы обеспечили, что у нас всегда будет возможность добавлять новые типы сообщений, которые не будут мешать использовать старые.

Это решение не обошлось без затрат. Парсеру JSON требуется несколько больше вычислительных ресурсов, чем очень простому и ограниченному парсеру, который он заменил, и, конечно, в новом парсере больше строк кода (что предполагает больше мест для возникновения дефектов). Кроме того, обычные анализаторы JSON для того, чтобы справиться с массивами переменной длины и со

словарями, которые описываются в JSON, требуют динамического распределения памяти, а динамическое распределение памяти является пресловутой причиной появления дефектов.

Мы справились с этими проблемами несколькими способами. Первым шагом было написание парсера на C для (достаточно) большого подмножества JSON, в котором используется исключительно статическое распределение памяти. Это потребовало принять некоторые незначительные ограничения, например, объекты в нашем диалекте JSON не могут содержать значение `null`, а массивы всегда имеют фиксированную максимальную длину. Приняв эти ограничения, мы смогли втиснуть парсер в 600 строк кода на языке C.

Затем мы построили полный набор юнит-тестов для парсера с тем, чтобы проверять отсутствие ошибок в операциях. Наконец, для встроенных систем с очень жесткими ограничениями, где накладные расходы для JSON могут быть слишком высокими, мы написали экспортный механизм с совместно используемой памятью, который позволяет обходиться без полного разбора и отправки сообщений JSON полностью в случае, если демон и его клиент имеют доступ к общей памяти.

JSON уже не используется только для веб-приложений. Мы считаем, что любой, кто проектирует протокол для приложения, должен рассматривать подход, похожий на используемый в проекте GPSD. Конечно, идея создания протокола поверх стандартного метапротокола не нова; поклонники XML пользуются ей в течение многих лет, и это имеет смысл для протоколов со сложной структурой документов. JSON обладает преимуществом за счет более низких накладных расходов, чем в случае с XML, и он лучше подходит при обходе массивов и структур записей.

7.9. Проектирование с минимизацией количества дефектов

Любое программное обеспечение, которое используется в навигационных системах и находится между пользователем и датчиком GPS или другим датчиком, определяющим месторасположение, потенциально является жизненно-важным, особенно в море или в воздухе. В навигационном программном обеспечении с открытым исходным кодом стараются уклониться от этой проблемы при помощи добавления соглашения об отказе от ответственности, в котором говорится: «Не полагайтесь на это, если это может поставить под угрозу жизнь».

Мы считаем, что подобные отказы бесполезны и опасны: бесполезны, поскольку системные интеграторы, весьма вероятно, относятся к ним формально и их игнорируют, и опасны, поскольку они способствуют тому, что разработчики опрометчиво считают, что дефекты в коде не будут иметь серьезных последствий, и что допустимо сглаживать острые углы в сфере обеспечения качества программ.

Разработчики проекта GPSD считают, что единственной приемлемой политикой является проект с нулевым количеством дефектов. Из-за того, что программное обеспечение остается сложным, мы этого еще не совсем достигли, но если учесть размер проекта GPSD, время работы над ним и его сложность, то мы подошли очень близко.

Нашей стратегией для этого является сочетание принципов архитектуры и политики кодирования, которые направлены на то, чтобы *исключить возможность появления дефектов в поставляемом коде*.

Один из важных принципов состоит в следующем: демон `gpsd` никогда не использует динамическое распределение памяти - нет ни `malloc`, ни `calloc`, и нет никаких обращений к функциям или библиотекам, в которых они требуются. Благодаря этому одним махом исчезает самая пресловутая причина дефектов, имеющая место при кодировании на языке C. У нас нет никаких утечек памяти и нет никаких ошибок двойных `malloc` или двойных `free` и никогда их не будет.

Нам это сходит с рук, поскольку все датчики, с которыми мы работаем, передают пакеты с относительно небольшой фиксированной максимальной длиной, а работа демона состоит в их обработке и отправке их клиентам почти без буферизации. Тем не менее, отказ от использования `malloc` требует дисциплины при кодировании и некоторых конструктивных компромиссов, некоторые из которых мы уже отметили при рассмотрении парсера JSON. Мы добровольно идем на эти затраты с тем, чтобы уменьшить количество дефектов.

Полезным побочным эффектом этой политики является то, что повышается эффективность чеккеров (checker) статического кода, например, `splint`, `cppcheck` и `Coverity`. Это ведет нас к другому крупному политическому выбору; мы интенсивно используем оба этих инструмента аудита кода, а также специально настроенный фреймворк регрессионного тестирования. Мы не знаем ни о каких других проектах, кроме GPSD, которые полностью аннотированы для использования `splint`, и сильно подозреваем, что ничего подобного еще не существует.

Строгая модульная архитектура GPSD помогает нам и здесь. Границы модулей служат в качестве точек отсечения, куда мы можем подсоединять специальные страховочные средства, и мы достаточно систематически это делаем. С помощью нашего обычного регрессионного тестирования проверяется все - от поведения плавающей точки на серверном оборудовании и до анализа JSON с тем, чтобы исправить сообщения о функционировании, поступающие из более чем семидесяти различных журналов датчиков.

Следует признать, что нам было несколько легче быть тщательными, чем многим другим приложениям, поскольку в демоне нет пользовательского интерфейса; окружающая его среда является простым набором последовательных потоков данных и ее относительно легко моделировать. Но, как и с отказом от использования `malloc`, фактическое использование этого преимущества требует правильного к нему отношения, что в частности означает, что нужно быть готовым тратить на проектирование и кодирование тестовых инструментальных средств и страховочных средств столько же времени, сколько мы тратим на создание кода. Это политика, которой, как мы думаем, могут и должны подражать другие проекты с открытым исходным кодом.

Когда я пишу эту статью (июль 2011 года), трекер ошибок проекта GPSD пуст. Он бывает пустым в течение недель, и на опыте появления сообщений об ошибках в прошлом, мы можем предполагать, что он будет оставаться таким же и далее. У нас в течение шести лет в коде не было критических ошибок. Когда у нас возникают ошибки, они, как правило, связаны к некоторым несущественными отсутствующими функциями или несоответствием спецификациям, что легко исправляется в течение нескольких минут.

Это не означает, что проект был непрерывной идиллией. Далее мы рассмотрим некоторые из наших ошибок ...

7.10. Усвоенные уроки

Проектировать программное обеспечение трудно; в нем обычно также есть ошибки и темные закоулки, и проект GPSD не стал исключением из этого правила. Самой большой ошибкой в истории этого проекта была разработка исходного протокола, который использовался перед использованием JSON для запросов и получения информации GPS. На то, чтобы от него отказаться, потребовались годы усилий и мы усвоили уроки, связанные как ошибочным первоначальным проектированием, так и с его исправлением.

С исходным протоколом были связаны две серьезные проблемы:

1. Плохая расширяемость. В нем использовались теги запросов и ответов, содержащие каждый одну букву без учета регистра. Так, например, запрос на получение долготы и широты был "P", а ответ выглядел, например, "P -75.32 40.05". К тому же, парсер интерпретировал

запрос, например, "PA", как запрос "P" с последующим запросом "A" (*altitude* - высота). Поскольку возможности демона постепенно расширялись, мы буквально были вытолкнуты из пространства команд.

2. Несоответствие между неявной моделью поведения датчиков, подразумеваемой в протоколе, и тем как датчики ведут себя на самом деле. Старый протокол состоял из запросов/ответов: посыпался запрос о местоположении (или высоты, или еще чего-нибудь), который возвращал ответ через некоторое время. На самом деле, как правило, не представляется возможным запрашивать ответ от датчиков GPS или других датчиков, относящихся к навигации; они выдают поток ответов, и лучшее, что запрос может сделать, это запросить кэш. Такое несоответствие поощряло неаккуратную обработку данных в приложениях: слишком часто они запрашивают данные о местоположении, не запрашивая время или иную проверочную информацию, касающуюся качества ответа, что на практике может легко привести к тому, что пользователю будут представлены устаревшие или неправильные данные.

Еще в 2006 году стало ясно, что старый дизайн протокола был несовершенным, но для того, чтобы разработать новый протокол, потребовалось почти три года проектных эскизов и фальшстартов. После этого переход занял два года, что стало причиной головной боли у разработчиков клиентских приложений. Стоимость этого перехода могла бы быть большей, если бы проект не поставлялся в виде библиотек клиентской стороны, которые изолировали пользователей от большинства деталей протокола, но вначале у нас не было достаточно хорошего API этих библиотек.

Если бы мы знали тогда, что мы знаем теперь, то протокол на основе JSON был бы внедрен на пять лет раньше, и в проекте интерфейса API клиентских библиотек потребовалось бы делать гораздо меньше изменений. Но есть ряд уроков которые можно усвоить только благодаря практике и эксперименту.

Есть по крайней мере две рекомендации по проектированию, которые нужно иметь ввиду для того, чтобы в будущих демонах сервисов избегать повторения ошибок:

1. Проектирование возможности расширения. Если в протоколе приложения вашего демона может не хватать пространства имен, как было в нашем старом протоколе, то вы сделали этот протокол неправильно. Переоценка краткосрочных затрат и недооценка долгосрочных преимуществ таких метапротоколов, как XML и JSON, являются ошибками, которые все еще остаются достаточно типичными.
2. Библиотеки клиентской стороны являются лучшим подходом, нежели предоставление доступа ко всем деталям протокола приложения. Внутренняя реализация библиотеки может быть адаптирована для многих версий протокола приложения, существенно снижая как сложность интерфейса, так и показатель наличия дефектов, в сравнении с альтернативным вариантом, когда каждый автор приложения должен разрабатывать специальное решение. Это различие выливается в гораздо меньшее количество сообщений об ошибках на трекере вашего проекта.

Одним из возможных ответов на наш упор на расширяемость, причем не только в протоколе приложений GPSD, но и в других частях архитектуры проекта, например, в интерфейс драйверов пакетов, является отказ от расширяемости как от нечего лишнего, что ведет изменению назначения проекта. Программисты Unix, прошедшие школу в традиции «делать что-то одно хорошо», могут спросить, действительно ли в 2011 году нужен больший набор команд `gpsd`, чем было в 2006 году, и почему `gpsd` теперь работает с датчики, которые не являются датчиками GPS, например, с магнитными компасами и морскими приемниками Marine AIS, и почему мы обдумываем такие возможности, как слежение за воздушными судами ADS-B.

Это справедливые вопросы. Мы можем получить ответ, если взглянем на фактическую сложность добавление нового типа устройства. По очень веским причинам, в том числе из-за относительно

низкого объема данных и высоких уровней электрических шумов, что исторически связано с последовательным подключением датчиков, почти все протоколы ответов датчиков GPS и других навигационных датчиков выглядят во многом одинаково: маленькие пакеты с проверочной контрольной суммой некоторого вида. Такие протоколы неудобно обрабатывать, но их действительно нетрудно отличать друг от друга и анализировать, а дополнительные затраты на добавление нового протокола, как правило, меньше, чем делать для каждого протокола что-то отдельное. Затраты даже на самые сложные из поддерживаемых нами протоколов с добавлением собственных генераторов ответов, например, для Marine AIS, составляют порядка 3 тыс. строк на каждый протокол. Драйвера плюс анализатор пакетов и связанные с ними генераторы ответов в формате JSON составляют приблизительно в общей сложности 18 тыс. строк кода.

Если это сравнивать с 43 тыс. строк кода всего проекта в целом, то мы видим, что большая часть затрат по сложности сосредоточена в GPSD на самом деле в коде фреймворка, окружающего драйверы, и (что важно) в инструментальных средствах тестирования и во фреймворке проверки правильности демона. Их дублирование привело бы к проекту намного большего размера, чем просто создание любого отдельного парсера пакетов. Так что написание проекта, эквивалентного GPSD, для протокола пакетов, который не обрабатывается в GPSD, потребовало бы гораздо больше работы, чем добавление еще одного драйвера и набора тестов для самого GPSD. Напротив, наиболее экономичный подход (и с наименьшим накапливаемым влиянием на показатель дефектов) представляет собой увеличение в GPSD количества драйверов пакетов для большого количества различных типов датчиков.

«То одно», ради чего проект GPSD разрабатывался так, чтобы он делал это хорошо, является работа с любым набором датчиков, которые передают пакеты с различными контрольными суммами. То, что выглядит как изменение назначения, является, на самом деле, предотвращением ситуации, к которой потребовалось бы писать много различных и дублирующих друг-друга демонов обработки. Вместо этого, разработчики приложений получили один относительно простой интерфейс API и выгоду благодаря нашей трудной экспертной победе в проектировании и тестировании большего количества типов датчиков.

То, что отличает GPSD от простой кучи возможностей нечеткого назначения, это не просто удача или черная магия, а грамотное применение известных лучших приемов разработки программного обеспечения. Отдача от них начинается с низкого уровня дефектов в настоящее время, и продолжается благодаря возможностью поддерживать новые функции без особых усилий или существенного влияния на уровень дефектности в будущем.

Возможно, самый важный урок, который мы получили для других проектов с открытым кодом, заключается в следующем: снижение дефектности асимптотически близко к нулю является сложной, но не невозможной задачей даже для такого широко внедряемого и разнообразного по назначению проекта, каким является проект GPSD. Этого можно достичь с помощью правильной архитектуры, хорошей практики кодирования, и действительными намерениями сосредоточиться на тестировании - и самым важным условием является дисциплина, которой надо следовать в этих трех направлениях.

8. Среда времени выполнения динамических языков и языки Iron

Языки Iron являются неформальной группой реализаций языков, у которых, в честь первого из них — языка IronPython, в названиях есть приставка «Iron» (железный — *прим.пер.*). У всех у них есть, по крайней мере, одна общая черта: они являются динамическими языками, ориентированными для использования в общей среде времени выполнения Common Language Runtime (CLR), которая более известна как .NET Framework, и они построены поверх динамической среды времен выполнения.

нения Dynamic Language Runtime (DLR). «CLR» является общим термином; платформа .NET Framework реализована фирмой Microsoft, также есть реализация с открытым исходным кодом - Mono. DLR представляет собой набор библиотек для CLR, которые обеспечивают гораздо лучшую поддержку динамических языков в среде CLR. Оба языка IronPython и IronRuby используются в нескольких десятках проектов с закрытым и с открытым исходным кодом, и оба находятся в стадии активной разработки; среда DLR, разработка которой началась как проект с открытым кодом, включена в качестве частей в проекты .NET Framework и Mono.

С точки зрения архитектуры языки IronPython и IronRuby и среда DLR являются одновременно как достаточно простыми, так и необычайно сложными. На самом высоком уровне их проекты похожи на многие другие реализации языков, с анализаторами и компиляторами, а также генераторами кода; но посмотрите на них поближе и начнут появляться интересные подробности: для того, чтобы сделать так, чтобы динамические языки выполнялись почти так же быстро, как статические языки на платформе, которая разработана для статических языков, используются точки вызовов, средства привязки, адаптивная компиляция и другие методы.

8.1. История

История языков Iron начинается в 2003 году. Джим Хаганин (Jim Hugunin) уже написал для виртуальной машины Java (JVM) реализацию языка Python, которая называлась Jython. В тот момент среда Common Language Runtime (CLR) для .NET Framework, тогда еще новая, рассматривалась некоторыми (кем именно, точно сказать не могу) как плохо подходящая для реализации таких динамических языков, как Python. После того, как Джим реализовал язык Python на JVM, ему стало интересно, насколько фирма Microsoft сделала платформу .NET, возможно, хуже, чем язык Java. В сентябре 2006 года он писал в блоге:

Я хотел понять, насколько сильно фирма Microsoft потеряет свое лицо, т. к. среда CLR была как платформа хуже для динамических языков, чем JVM. Мой план состоял в том, чтобы потратить пару недель для сборки прототипа реализации языка Python на CLR, а затем использовать эту работу для того чтобы написать короткую содержательную статью под названием «Почему среда CLR является ужасной платформой для динамических языков». Пока я работал над прототипом, мои планы быстро изменились, поскольку обнаружил, что язык Python может работать в среде CLR - во многих случаях заметно быстрее, чем реализация на основе языка C. Для стандартного бенчмарка pystone язык IronPython в среде CLR работал приблизительно в 1,7 раза быстрее, чем реализация на основе языка C.

Часть «Iron» («железный»), используемая в имени, стала обыгрыванием названия компании Want of a Nail Software, в которой в то время работал Джим. (*Прим.пер.: Want of a Nail Software означает программы, выполняющие некоторые незначительные операции, которые, в общей совокупности, могут оказывать достаточно большое воздействие, т. е. что-то вроде «эффекта бабочки».*)

Вскоре после этого Джим был нанят фирмой Microsoft для того, чтобы сделать платформу .NET более подходящей для динамических языков. Джим (и несколько других разработчиков) создал среду DLR при помощи выделения языково-нейтральных частей оригинального кода реализации IronPython в отдельный код (факторинга). Для того, чтобы создать общее ядро реализации динамических языков платформы .NET, была разработана среда DLR и она было основной новой особенностью платформы .NET 4.

В тот момент, когда была анонсирована среда DLR (апрель 2007), фирма Microsoft для того, чтобы продемонстрировать приспособляемость среды DLR к различным языкам, также объявила, что в добавок к новой версии языка IronPython, собранного на базе среды DLR (IronPython 2.0), также на базе среды DLR будет разрабатываться язык IronRuby. В октябре 2010 года Microsoft прекратила разработку языков IronPython и IronRuby, и они стали независимыми проектами с открытым исходным кодом. Интеграция с динамическими языками, использующими среду DLR, также рас-

сматривалась как основная особенность языков C# и Visual Basic, использующих новое ключевое слово (`dynamic`), что позволило этим языкам легко обращаться к любому языку в среде DLR или к любому другому динамическому источнику данных. Среда CLR уже была хорошей платформой для реализации статических языков, а среда DLR делает динамические языки основными игроками платформы.

К числу других реализаций языков, которыми не занималась фирма Microsoft, но в которых также используется среда DLR, относятся [IronScheme](#) и [IronJS](#). Кроме того, в PowerShell v3 фирмы Microsoft также будет использована среда DLR вместо ее собственной системы динамических объектов.

8.2. Принципы реализации среды времени выполнения динамических языков

Среда CLR создана с учетом использования статически-типовизированных языков; то, что типы данных известны, используется в среде времени выполнения очень глубоко, и одним из ключевых условий является то, что эти типы не должны изменяться, — что переменная никогда не изменит свой тип или, что в типе никогда нет будет никаких полей или членов типа, которые добавляются или удаляются во время работы программы. Это нормально для таких языков, как C# или Java, но в динамических языках эти правила, по определению, не выполняются. В среде CLR также предоставлена единая система объектов статических типов, что означает, что любой язык платформы .NET может вызывать объекты, написанные на любом другом языке платформы .NET, причем без дополнительных усилий.

Без среды DLR в каждом динамическом языке потребовалось бы создавать свою собственную модель объектов; разные динамические языки не могли бы обращаться к объектам других динамических языков, а C# не мог бы одинаковым образом обрабатывать языки IronPython и IronRuby. Поэтому сердцем среды DLR является стандартный способ реализации *динамических объектов* и, при этом, с помощью *средств привязок* (*binders*) все еще есть возможность настраивать свойства объектов для каждого конкретного языка. Также есть механизм, называющийся *кэшированием точек вызовов* (*call-site caching*), который обеспечивает выполнение динамических операций настолько быстро, насколько это возможно, и набор классов для создания *деревьев выражений* (*expression trees*), которые позволяют сохранять код в виде данных и легко им манипулировать.

В среде CLR также предоставляется ряд других возможностей, которые полезны для динамических языков, в том числе интеллектуальный сборщик мусора; компилятор Just-in-Time (JIT), преобразующий во время выполнения программы на байт-коде в обобщенный промежуточный язык Common Intermediate Language (IL), который компиляторы платформы .NET предобразуют в машинный код; система интроспекции времени выполнения, позволяющая динамическим языкам вызывать объекты, написанные на любом статическом языке; и, наконец, динамические методы (также известные как легковесная генерация кода), которые позволяют во время выполнения программы генерировать код, а затем исполнять его с затратами, лишь чуть-чуть превышающими затраты на статические методы вызовов (в виртуальной машине JVM языка Java 7 аналогичный механизм реализуется с помощью `>invokedynamic<`).

Благодаря такой архитектуре среды DLR языки, такие как IronPython и IronRuby, могут вызвать объекты друг друга (а также любого другого языка DLR), поскольку у них общая динамическую модель объектов. Поддержка такой объектной модели была также добавлена в язык C# 4 (с помощью ключевого слова `dynamic`) и в Visual Basic 10 (в дополнение к методу «позднего связывания», уже существующему в VB), так что в них также можно выполнять динамические вызовы объектов. Таким образом среда DLR делает динамические языки основными игроками платформы .NET.

Интересно то, что среда DLR полностью реализована в виде набора библиотек и может также быть встроена и работать на платформе .NET 2.0. Для ее реализации в среде CLR не требуется делать никаких изменений.

8.3. Особенности реализации языков

Реализация каждого языка содержит в себе два основных этапа — синтаксический анализ или разбор (представление языка) и генерацию кода (движок). В среде DLR в каждом языке реализуется свое собственное представление языка, которое содержит синтаксический анализатор языка и генератор синтаксического дерева; среда DLR предоставляет общий движок, который использует деревья выражений для того, чтобы создать код на промежуточном языке Intermediate Language (IL) для его использования в среде CLR; среда CLR передает код на языке IL в компилятор JIT (Just-In-Time — компиляция «на лету»), с помощью которого создается машинный код, предназначенный для выполнения в процессоре. Код, который определяется во время выполнения (и работает с использованием команды `eval`) обрабатывается аналогичным образом, за исключением лишь того, что все это происходит исключительно в точке вызова `eval`, а не когда загружается файл.

Есть несколько различных способов реализации ключевых элементов представления языка, и хотя языки IronPython и IronRuby очень похожи друг на друга (в конце концов, они разрабатывались бок о бок,) они различаются в нескольких ключевых аспектах. Оба языка IronPython и IronRuby имеют довольно стандартные структуры синтаксических анализаторов — оба они используют *tokenizer* (также известный как *лексический анализатор*), который разделяет текст на лексемы, а затем *синтаксический анализатор / парсер* преобразует эти лексемы в *абстрактное синтаксическое дерево AST (abstract syntax tree)*, которое представляет собой программу. Однако реализация этих компонентов в этих языках совершенно разная.

8.4. Синтаксический анализ

Лексический анализатор языка IronPython находится в классе `IronPython.Compiler.Tokenizer`, а парсер — в классе `IronPython.Compiler.Parser`. Лексический анализатор представляет собой конечный автомат, написанный вручную, который распознает ключевые слова, операторы и имена языка Python и создает соответствующие лексемы. Каждая лексема сопровождается дополнительной информацией (например, значение константы или имени), а также для того, чтобы было проще вести отладку, указывается, где в исходном была найдена лексема. Затем парсер берет этот набор лексем и сравнивает их с грамматикой языка Python с тем, чтобы поверить, соответствует ли этот набор правильным конструкциям языка Python.

Синтаксический анализатор языка IronPython представляет собой *синтаксический анализатор рекурсивного спуска вида LL(1) (LL(1) recursive descent parser)*. Анализатор будет анализировать входящую лексему и, если лексема допустима, вызвать функцию, либо будет возвращать ошибку, если это не так. Анализатор рекурсивного спуска собран из набора взаимно рекурсивных функций; в конечном итоге эти функции реализуют конечный автомат, в котором каждая новая лексема вызывает переход между состояниями. Точно также, как и лексический анализатор, парсер языка IronPython написан вручную.

С другой стороны в языке IronRuby используются лексический и синтаксический анализаторы, сгенерированные генератором анализаторов Gardens Point Parser Generator (GPPG). Парсер описан в файле `Parser.y` (`Languages/Ruby/Ruby/Compiler/Parser/Parser.y`), являющимся файлом в формате yacc, в котором на высоком уровне с помощью правил (rules), задающих грамматику, описана грамматика языка IronRuby. После этого генератор GPPG берет файл `Parser.y` и строит фактически используемые функции и таблицы парсера в результате получается парсер вида LALR(1), использующий в своей работе таблицы. Созданные таблицы представляют собой дли-

ные массивы целых чисел, где каждое целое представляет собой состояние; в таблице по текущему состоянию и текущей лексеме определяется, какое состояние будет следующим. Анализатор рекурсивного спуска языка IronPython читается довольно легко, тогда как сгенерированный парсер языка IronRuby прочитать невозможно. Таблица переходов огромна (540 различных состояний и более 45 000 переходов), и вручную ее изменить практически невозможно.

В конечном счете, это компромисс разработки — парсер языка IronPython достаточно прост с тем, чтобы его можно было изменять вручную, но достаточно сложный, поскольку он скрывает структуру языка. Парсер языка IronRuby, с другой стороны, позволяет гораздо легче понять структуру языка, находящегося в файле `Parser.y`, но теперь он зависит от стороннего инструментального средства, в котором применяется специальный (хотя и известный) язык конкретной прикладной области, в котором могут быть свои собственные ошибки или особенности. В данном случае команда разработчиков языка IronPython не хотела быть зависимой от внешнего инструментального средства, тогда как команда разработчиков языка IronRuby не возражала против этого.

Однако совершенно ясно, насколько на каждом этапе для анализа важны конечные автоматы. Для любой задачи синтаксического анализа независимо от того, насколько она будет простой, конечный автомат всегда выдает правильный ответ.

Прим.пер.: В данном случае приводится описание лишь конкретных реализаций. В общем случае возможностей конченого автомата может оказаться недостаточной для реализации грамматик вида LL(1) и LALR(1). Что же касается правильности ответов, выдаваемых конечными автоматами, то они ровно настолько правильные, насколько правильно построена таблица переходов конченого автомата, причем это не зависит от того, будет ли эта таблица построена вручную в виде вызовов взаимно рекурсивных функций, либо она будет описана в виде грамматики, которая затем преобразуется в длинные массивы целых чисел.

Результатом работы синтаксического анализатора для любого языка является абстрактное синтакическое дерево (AST). В нем на высоком уровне описывается структура программы, при этом каждый узел отображается непосредственно в конструкции языка — оператор или выражение. С этими деревьями можно манипулировать во время выполнения, часто для того, чтобы перед компиляцией оптимизировать программу. Впрочем, дерево AST некоторого языка связана с конкретным языком, а среда DLR должна работать с деревьями, в которых нет никакого конструкций конкретного языка, а содержатся только общие конструкции.

8.5. Деревья выражений

Дерево выражений (*expression tree*) также является представлением программы, с которым можно манипулировать во время выполнения, но в более низкогоуровневом представлении, независимым от языка. На платформе .NET, типы узлов находятся в пространстве имен `System.Linq.Expressions`, и все типы узлов являются производными от абстрактного класса `Expression`. Пространство имен является историческим артефактом; первоначально в .NET 3.5 для реализации языка интегрированных запросов LINQ - Language-Integrated Query были добавлены деревья выражений, а затем они были расширены деревьями выражений DLR. Эти деревья выражений используются не только в выражениях, хотя также есть типы узлов для инструкций `if`, блоков `try` и циклов; в некоторых языках (в Ruby, например) есть выражения и нет инструкций.

Есть узлы, которые покрывают практически все функции, которые могут потребоваться в языке программирования. Впрочем, есть тенденция определять их на достаточно низком уровне; вместо того, чтобы иметь узлы `ForExpression`, `WhileExpression` и т.д., существует один узел `LoopExpression`, которые в сочетании с узлом `GotoExpression` может описывать циклы любого типа. Для того, чтобы описывать язык на более высоком уровне, в языках можно определять свои собственные типы узлов с помощью их наследования из `Expression` и переопределения метода `Reduce()`, который возвращает другое дерево выражений. В IronPython, дерево разбора также явля-

ется деревом выражений DLR, но в нем есть множество специальных узлов, которые обычно не-понятны для DLR (например, `ForStatement`). Эти специальные узлы могут быть приведены к деревьям выражений, которые хорошо понятны в DLR (например, сочетание `LoopExpressions` и `GotoExpressions`). Эти специальные узлы выражений могут приводиться к другим специальным узлам выражений, поэтому приведение продолжается рекурсивно до тех пор, пока не останутся только узлы DLR. Одно из ключевых различий между IronPython и IronRuby состоит в том, что абстрактное синтаксическое дерево (AST) языка IronPython также является деревом выражения, а в языке IronRuby - не является. Вместо этого, прежде, чем произойдет переход на следующий этап, дерево AST языка IronRuby преобразуется в дерево выражений. На самом деле неясно, насколько полезно то, что дерево AST также может быть деревом выражений, поэтому оно не было реализовано таким образом в языке IronRuby.

Для каждого типа узла известно, как его можно сократить, причем обычно сокращение можно выполнить единственным образом. Для преобразований, которые выполняются над кодом, не находящимся в дереве — например, сворачивание констант или реализация генераторов Python в IronPython — используется подкласс класса `Expression`. В классе `ExpressionVisitor` есть метод `Visit()`, который вызывает метод `Accept()` класса `Expression`, а в подклассе `Expression` метод `Accept()` переопределяется таким образом, чтобы вызвать конкретный метод `Visit()` класса `ExpressionVisitor`, например, `VisitBinary()`. Это классическая реализация *паттерна Visitor*, предложенного Gamma и другими - есть фиксированный набор типов узлов, которые можно посетить, и есть бесконечное количество операций, которые над ними могут быть выполнены. Когда при разборе выражения посещается узел, то, как правило, рекурсивно посещаются его потомки, а также потомки его потомков и так далее вниз по дереву. Однако метод `ExpressionVisitor` не может в действительности изменить дерево при его просмотре, поскольку деревья выражений являются немутируемыми (неизменяемыми). Если необходимо изменить узел (например, удалить потомков), то вместо этого следует создать новый узел, который заменит старый узел, а также всех его предков.

После того, как дерево выражений будет создано, сокращено и преобразовано к паттерну *Visitor*, его, в конце концов, нужно будет выполнить. Хотя деревья выражений могут быть откомпилированы непосредственно в код IL, в IronPython и IronRuby происходит их первоначальное перенаправление в интерпретатор, т. к. компиляция непосредственно в IL является слишком дорогой для кода, который, возможно, будет выполнен всего-лишь несколько раз.

8.6. Интерпретация и компиляция

Одним из недостатков использования компилятора JIT, например, применяемого на платформе .NET, является то, что ему при запуске требуется время для того, чтобы преобразовать байт-код IL в машинный код, который может исполняться процессор. При JIT-компиляции создается код, который работает намного быстрее, чем при работе через интерпретатор, но в зависимости от того, что требуется сделать, затраты на запуск могут оказаться непомерно высокими. К примеру, долгоживущий серверный процесс, например, веб-приложение, выигрывает от использования JIT, поскольку время запуска почти не имеет значения, а время, необходимое для каждого запроса, является решающим, причем один и тот же код выполняется повторно. С другой стороны, для программы, которая запускается часто, но только на короткий промежуток времени, например, для клиентской программы Mercurial, работающей из командной строки, было бы лучше иметь небольшое время запуска, поскольку в ней, скорее всего, каждый фрагмент кода выполняется только один раз, а то, что код, созданный компилятором JIT, работает быстрее, не скомпенсирует того, что на запуск потребуется гораздо больше времени.

Платформа .NET не может выполнять код IL непосредственно; она всегда использует компилятор JIT для компиляции в машинный код, а на это требуется время. В частности, время запуска программ является одной из слабых сторон фреймворка .NET, поскольку компилятору JIT нужно откомпилировать большую часть кода. Хотя есть способы, которые позволяют избегать подобных

затрат для статических программ платформы .NET (генерация нативных образов - Native Image Generation, или [NGEN](#)), для динамических программ они не работают. Вместо того, чтобы выполнять компиляцию всегда непосредственно в код IL, в языках IronRuby и IronPython будут использоваться их собственные интерпретаторы (находится в `Microsoft.Scripting.Interpreter`), которые работают не так быстро, как код, откомпилированный с помощью JIT, но при запуске затрачивают гораздо меньше времени. Интерпретатор также полезен в ситуациях, когда не допускается динамическая генерация кода, например, на мобильных платформах; в противном случае языки DLR вообще не смогут работать.

Перед тем, как начнется выполнение, все дерево выражений должно быть преобразовано в функцию так, что его можно было выполнить. В языках DLR функции представлены в виде узлов `LambdaExpression`. В большинстве языков функция `lambda` является анонимной функцией, а в языках DLR концепция именования не используется; все функции являются анонимными. Узел `LambdaExpression` уникален тем, что это единственный тип узла, который может быть преобразован в делегата `delegate`, т. е. в такую сущность, которая на платформе .NET с использованием метода `Compile()` вызывает функции первого порядка. Делегат похож на указатель функций в языке С - это просто дескриптор куска кода, который можно вызывать.

Сначала дерево выражения помещается в узел `LightLambdaExpression`, из которого можно также создать делегата, который может быть выполнен; однако вместо того, чтобы генерировать код на языке IL (для которого затем нужно будет вызывать компилятор JIT), дерево выражений компилируется в список инструкций, которые затем выполняются на простой виртуальной машине интерпретатора. Интерпретатор является простым стековым интерпретатором; инструкции берут значения из стека, выполняют операцию, а затем помещают результат обратно в стек. Каждая команда представляет собой экземпляр класса, производного от класса

`Microsoft.Scripting.Interpreter.Instruction` (например, `AddInstruction` или `BranchTrueInstruction`), в котором есть свойства, описывающие, сколько элементов выбирается из стека, сколько из них используется, и метод `Run()`, который выполняет инструкцию, выбирая значения из стека, помещая результат в стек и возвращая смещение для следующей инструкции. Интерпретатор берет список инструкций и выполняет их одну за другой, переходя вперед или назад в зависимости от значения, возвращаемого методом `Run()`.

Как только некоторая часть кода будет выполнена определенное количество раз, она будет с помощью вызова метода `LightLambdaExpression.Reduce()` преобразована в полный вариант узла `LambdaExpression`, затем будет выполнена компиляция в делегата `DynamicMethod` (в фоновом потоке с целью определенного распараллеливания), и точка вызова старого делегата будут заменены новым вызовом, более быстрым. Это значительно снижает стоимость выполнения функций, вызываемых всего-лишь несколько раз, например, функция `main` программы, тогда как функции, к которым происходит регулярное обращение, работают настолько быстро, насколько это возможно. По умолчанию, порог компиляции установлен равным 32 кратному исполнению кода, но это значение можно изменять с помощью параметра командной строки или из хост-программы; также можно полностью отключить либо компиляцию, либо интерпретацию.

Независимо от того, работает ли интерпретатор, или происходит компиляция в код IL, компилятор дерева выражений не выполняет непосредственное преобразование в операции языка. Вместо этого компилятор создает для каждой операции, которая может быть динамической (что бывает практически всегда), точку вызова. Благодаря таким точкам вызовов можно использовать динамические объекты, а код сохранит высокую производительность.

8.7. Точки динамических вызовов

В статическом языке платформы .NET все решения о том, какой код должен быть вызван, делаются на этапе компиляции. Например, рассмотрим следующую строку кода на языке C#:

```
var z = x + y;
```

Компилятору известны типы переменных 'x' и 'y' и то, можно ли их складывать. Компилятор, основываясь исключительно на статической информации, в которой есть все о типах переменных, может создать правильный код, необходимый для обработки перегружаемых операторов, преобразования типов и всего того, что может понадобиться для создания кода, который будет работать должным образом. Теперь рассмотрим следующую строку кода на языке Python:

```
z = x + y
```

Компилятор языка IronPython, когда он столкнется с такой строкой, понятия не имеет, что можно делать, поскольку ему не известны типы у x и y, и даже если они ему известны, то во время выполнения возможность складывать x и y может так или иначе измениться. В принципе, это узнать можно, но ни в языке IronRuby, ни в языке IronPython нет наследования типов. Вместо того, чтобы создавать код IL для сложения чисел, IronPython создает *точку вызова (call site)*, для которой во время выполнения будет определена вся информация.

Точка вызова является местом выполнения операции, которая будет определена на этапе исполнения программы; точки вызова реализуются как экземпляры класса `System.Runtime.CompilerServices.CallSite`. В динамических языках, например, в Python или Ruby, почти для каждой операции есть динамический компонент; такие динамические операции представлены в деревьях выражений как узлы `DynamicExpression` и компилятор дерева выражений знает, как их преобразовывать в точки вызова. Когда создается точка вызова, то еще не известно, как требуется выполнять операцию; но точка вызова будет создана с экземпляром надлежащего *средства привязки к точке вызова (call site binder)*, специального для каждого для используемого языка и содержащего всю необходимую информацию о том, как выполнять операцию.

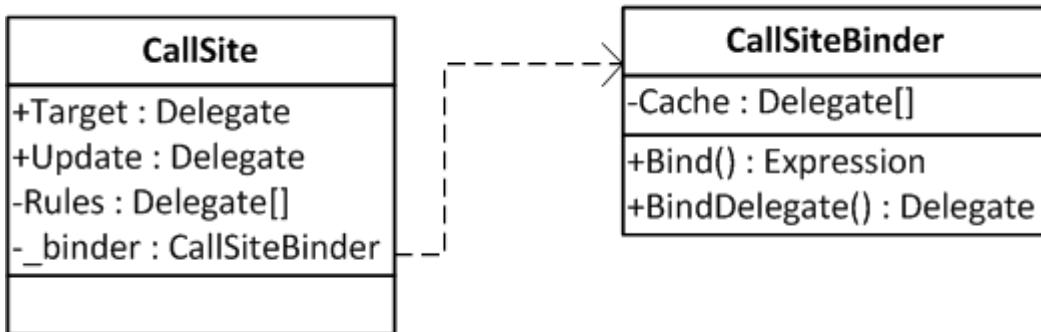


Рис.8.1: Диаграмма класса CallSite

В каждом языке для каждой операции будут свои собственные средства привязки к точкам вызова; причем этим средствам часто известно много различных способов выполнения операции в зависимости от того, какие в точке вызова используются аргументы. Однако, генерация таких правил дорогостоящая (в частности, компиляция их в виде делегата выполнения, в котором есть обращения к компилятору .NET JIT), поэтому в точке вызова есть многоуровневый *кэш точки вызова (call site cache)*, где уже созданные правила сохраняются для последующего использования.

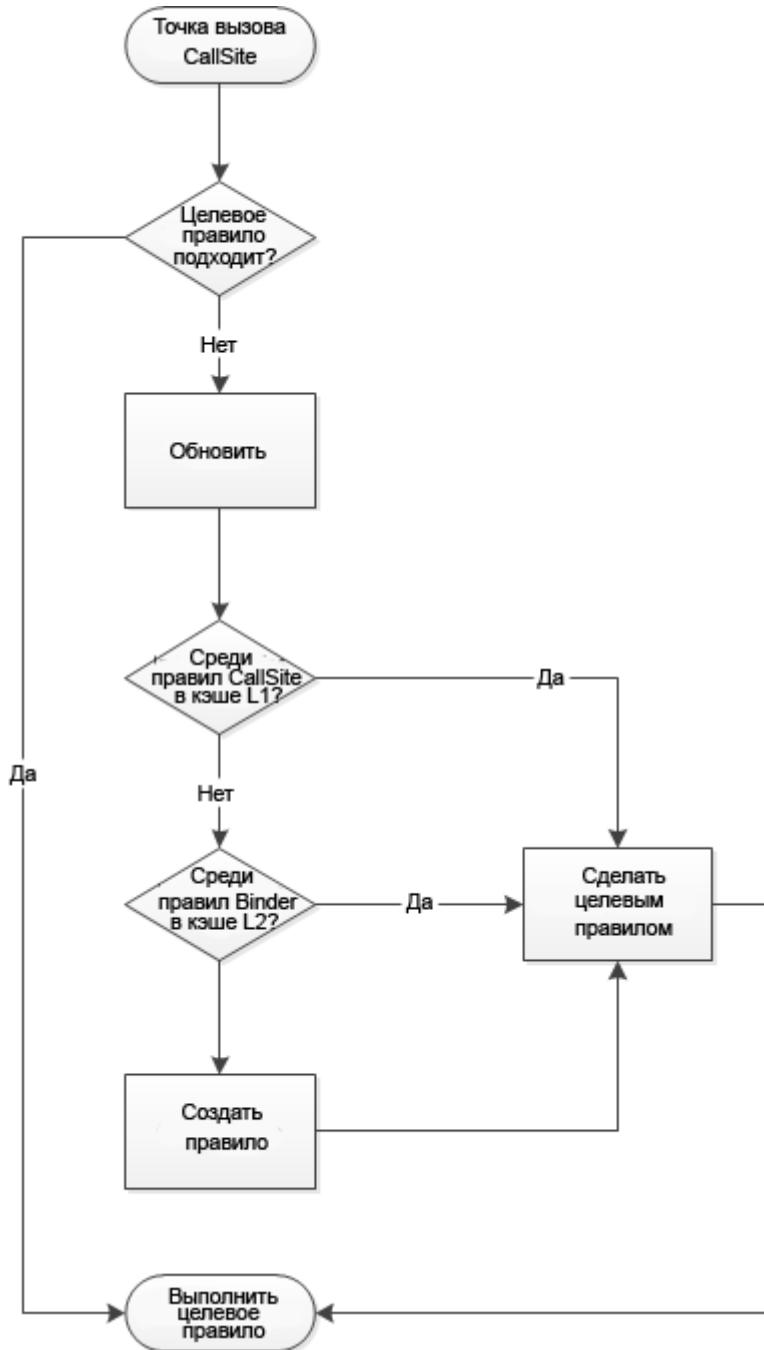


Рис.8.2: Блок-схема CallSite

Первым уровнем L0 является свойство `CallSite.Target` самого экземпляра точки вызова. В нем хранятся правила, которые использовались для этой точки вызова совсем недавно; для подавляющего количества точек вызова это все, что когда-либо потребуется, поскольку они всегда вызываются только с одним набором типов аргументов. Точка вызова также имеет еще один кэш L1, в котором хранятся еще 10 правил. Если свойство `Target` не подходит для данного вызова (например, если типы аргументов разные), то в точке вызова сначала проверяется кэш с правилами для того, чтобы выяснить, может ли надлежащий делегат создан из предыдущего вызова, и можно ли повторно использовать это правило, а не создавать новое.

Сохранение правил в кэше экономит время, поскольку время, необходимое для компиляции нового правила, значительно больше по сравнению со временем, которое нужно для проверки существующих правил. Конкретно говоря, чтобы выполнить проверку типа переменной, который является наиболее распространенным типом предиката правил, на платформе .NET затрачивается около 10 нс (проверка бинарной функции занимает 20 нсек, и т.д.). С другой стороны компиляция простого метода, который складывает пару чисел, занимает приблизительно 80 мкsec или на три по-

рядка больше. Размер кэша ограничивается с тем, чтобы предотвратить излишнюю трату памяти на запоминание каждого правила, которое применяется в точке вызова; для простого сложения на каждый вариант требуется около 1 Кб памяти. Однако, профилирование показало, что для очень небольшого количества точек вызова когда-либо используется более 10 вариантов.

Наконец, есть кэш L2, в котором хранится экземпляр самого средства привязки. Экземпляр привязки, который ассоциирован с точкой вызова, может хранить некоторую дополнительную информацию о том, что делает эта конкретная точка вызова, но, в любом случае, большая часть точек вызовов не будут уникальными и для них можно совместно использовать один и тот же экземпляр средства привязки. Например, в языке Python, базовые правила, используемые для сложения, будут во всей программе одинаковыми; они зависят от двух типов данных, находящихся по обе стороны операции + - и все. Все операции сложения в программе могут пользоваться одним и тем же средством привязки, и, если не удается воспользоваться обоими кэшами L0 и L1, то в кэше L2 можно обнаружить гораздо больше правил (128), которые использовались последними во всей программе. Даже если это первое выполнение точки вызова, есть достаточно большой шанс найти соответствующее правило в кэше L2. Для того, чтобы этот метод работал наиболее эффективно, и в IronPython и в IronRuby предлагается набор канонических экземпляров средств привязки, которые применяются для обычных операций, таких как сложение.

Если кэш L2 использован не будет, то средство привязки запросит создать *реализацию* (*implementation*) для точки вызова, в которой учтены типы аргументов (и, возможно, даже их значения). Если в приведенном выше примере x и y имеют тип double (или другой нативный тип), то реализацией будет просто приведение их к типу double и вызов инструкции сложения add на языке IL. Средство привязки также создает тест, который проверяет аргументы и гарантирует, что они допустимы для этой реализации. Реализация и тест вместе образуют правило. В большинстве случаев, как реализация, так и тест создаются и хранятся в виде деревьев выражений. Однако инфраструктура точек вызовов не зависит от деревьев выражений; ее можно использовать отдельно с делегатами.

Если деревья выражений записать на языке C#, то код будет похож на следующий:

```
if(x is double && y is double) {           // проверяется, является ли тип переменных типом double
    return (double)x + (double)y;           // если тип double, то происходит выполнение
}
return site.Update(site, x, y);             // если тип не double, то ищется/создается еще одно
                                            // правило для типов этих переменных
```

Затем средство привязки создает из деревьев выражений *делегата* (*delegate*), что означает, что правило компилируется в язык IL, а затем - в машинный код. В случае сложения двух чисел, это, вероятно, будет быстрая проверка типов, а затем - машинная команда сложения чисел. Даже с учетом всех используемых приемов, конечный результат будет лишь чуть-чуть медленнее, чем статический код. В языках IronPython и IronRuby также есть набор предварительно скомпилированных правил для обычных операций, таких как сложение примитивных типов, что позволяет экономить время, поскольку их не требуется создавать во время выполнения, но за это приходится заплатить некоторым дополнительным пространством на диске.

8.8. Протокол метаобъектов

Кроме того, инфраструктура языков, которая является другой ключевой частью среды DLR, дает возможность языку (основному языку или *хост-языку*) осуществлять динамические вызовы объектов, определенных на другом языке (*языке исходного кода объекта*). Чтобы это было возможно, среда DLR должна понимать, какие операции являются допустимыми над на объектом, независимо от того, на каком языке он был написан. В языках Python и Ruby есть довольно похожие моде-

ли объектов, но язык JavaScript имеет радикально другую систему типов, базирующихся на прототипах (в отличие от системы, базирующихся на классах). Вместо того, чтобы пытаться объединять различные системы типов, среда DLR обрабатывает их так, как если бы все они базировались на принципах *передачи сообщений* (*message passing*) в стиле языка Smalltalk.

В объектно-ориентированной системе, базирующейся на передаче сообщений, объекты посылают сообщения другим объектам (как правило, с параметрами), а затем объект может в качестве результата возвращать другой объект. Таким образом, хотя в каждом языке есть свое собственное понимание о том, что может представлять собой объект, они все почти всегда рассматриваться как эквивалентные за счет вызовов методов просмотра, которые передаются в виде сообщений между объектами. Конечно, даже статические объектно-ориентированные языки подходят к этой модели с некоторой натяжкой; отличие динамических языков состоит в том, что вызываемый метод не должны быть известен во время компиляции, или может даже вообще не существовать в объекте (например, метод `method_missing` языка Ruby), а у целевого объекта обычно есть возможность, если это потребуется, перехватить сообщение и обработать его по-своему (например, метод `__getattr__` языка Python).

В среде DLR определяются следующие сообщения:

- `{Get|Set|Delete}Member`: операции для манипулирования с членами объекта
- `{Get|Set|Delete}Index`: операций для индексированных объектов (например, массивов или словарей)
- `Invoke, InvokeMember`: вызов объекта или члена объекта
- `CreateInstance`: создание экземпляра объекта
- `Convert`: преобразование объекта из одного типа в другой
- `UnaryOperation, BinaryOperation`: выполнение операций с использованием операторов, таких как получение обратного значения (!) или сложение (+)

Этих операций, если взять их вместе, должно быть достаточно для реализации почти любой объектной модели языка.

Поскольку среда CLR наследует статическую типизацию, объекты динамических языков все еще необходимо выражать с помощью статических классов. Обычно это происходит с помощью статического класса, например, `PythonObject`, а реальные объекты языка Python являются экземплярами этого класса или его подклассов. Чтобы обеспечить возможность совместного использования и хорошую производительность, в среде DLR применяется гораздо более сложный механизм. Вместо того, чтобы иметь дело с объектами конкретных языков, в среде DLR используются *метаобъекты* (*meta-objects*), которые являются подклассами класса

`System.Dynamic.DynamicMetaObject` и имеют методы для обработки всех перечисленных выше видов сообщений. В каждом языке есть свой собственный подкласс `DynamicMetaObject`, реализующий модель объектов языка, например, класс `MetaPythonObject` в языке IronPython. В метаklassах также есть соответствующие конкретные классы, реализующие интерфейс `System.Dynamic.IDynamicMetaObjectProtocol`, в котором определено, как в среде DLR идентифицируются динамические объекты.

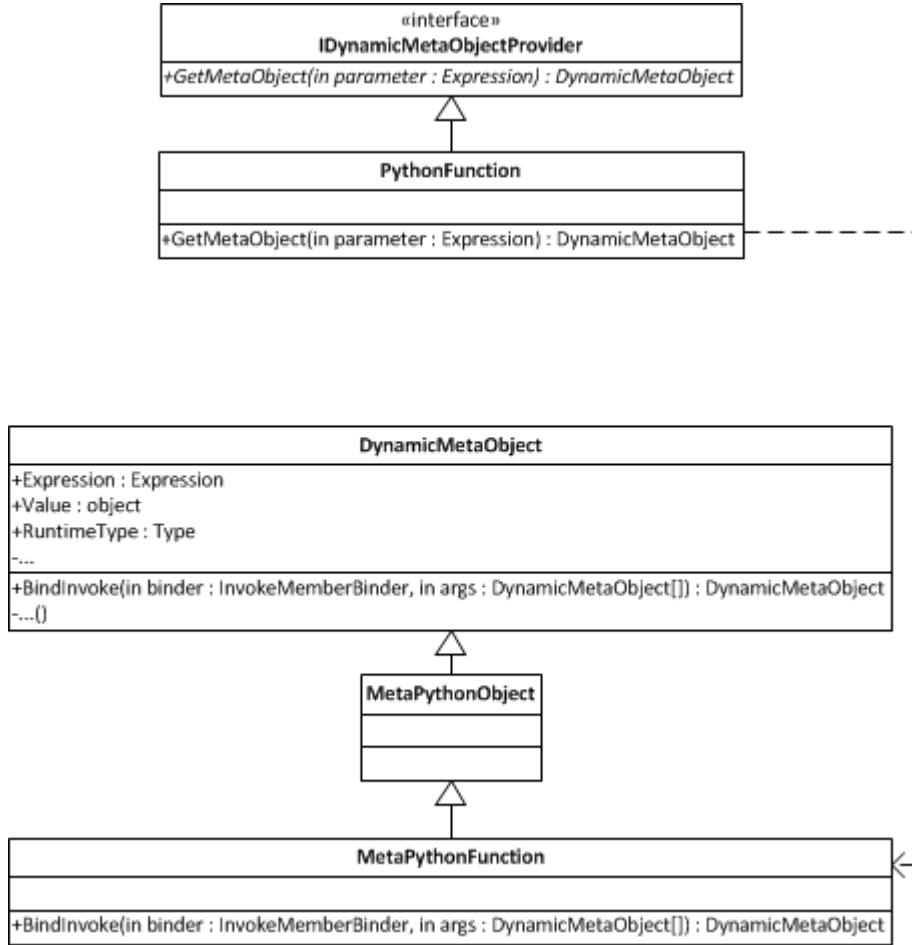


Рис.8.3: Диаграмма класса IDMOP

Из класса, в котором реализован протокол `IDynamicMetaObjectProtocol`, среда DLR с помощью вызова метода `GetMetaObject()` может получить объект `DynamicMetaObject`. Этот объект `DynamicMetaObject` будет предоставлен языком и будет реализован с помощью функций привязок, которые требуются самому объекту. У каждого объекта `DynamicMetaObject` также есть значение и тип, если таковые есть в объекте, лежащем в его основе. Наконец, в объекте `DynamicMetaObject` есть дерево выражений, в котором все еще хранятся точки вызовов и указаны ограничения на каждое выражение, аналогичные механизмам привязок точек вызова.

Когда среда DLR компилирует обращение к методу класса, определяемого пользователем, то сначала создается точка вызова (т.е. экземпляр класса `CallSite`). В точке вызова инициируется процесс привязки так, как он описан выше в разделе «Точки динамических вызовов», в результате чего он в конечном итоге происходит вызов метода `GetMetaObject()` в экземпляре класса `OldInstance`, который возвращает объект `MetaOldInstance`. В языке Python есть классы как старого, так и нового вида, но в данном случае это неважно. Затем вызывается средство привязки (а именно `PythonGetMemberBinder.Bind()`),зывающее, в свою очередь, метод `MetaOldInstance.BindGetMember()`; он возвращает новый объект `DynamicMetaObject`, который представляет собой новый механизм поиска имени метода объекта. После этого вызывается другое средство привязки `PythonInvokeBinder.Bind()`, в котором вызывается метод `MetaOldInstance.BindInvoke()`, представляющий собой обертку первого объекта `DynamicMetaObject` с новым способом вызова искомого метода. Здесь присутствует исходный объект, дерево выражений, используемого для поиска имени метода, и объекты `DynamicMetaObject`, представляющие собой аргументы, используемые в методе.

Как только в выражении будет построен объект `DynamicMetaObject`, его дерево выражений и ограничения будут использованы для создания делегата, затем возвращаемого в точку вызова, кото-

рая инициировала процесс привязки. С этого момента код можно сохранять в кэшах точки вызова, что позволяет выполнять операции над объектами также же быстро, как выполняются другие динамические вызовы, и почти также быстро, как выполняются статические вызовы.

Хост-языки, в которых требуется выполнять динамические операции для динамических языков, должны создавать соответствующие привязки из метапривязки `DynamicMetaObjectBinder`. `DynamicMetaObjectBinder` прежде, чем возвращаться к семантике привязок хост-языка, сначала запросит целевой объект, к которому привязывается операция (с помощью вызова метода `GetMetaObject()` и выполнения процесса связывания, описанного выше). В результате, если объект языка IronRuby будет доступен из программы IronPython, то привязка сначала будет происходить в семантике языка Ruby (целевой язык); а если этого сделать не удастся, то привязка `DynamicMetaObjectBinder` вернется к семантике языка Python (хост-язык). Если объект, для которого делается привязка, не динамический (то есть, в нем не реализован провайдер `IDynamicMetaObjectProvider`), как, например, классы из базовой библиотеки платформы .NET, то для доступа к нему с семантикой хост-языка используется механизм .NET reflection.

В языкам предоставлена относительная свобода реализации этого механизма; реализация `PythonInvokeBinder` в языке IronPython не выводится из `InvokeBinder`, т.к. для объектов языка Python нужно выполнять некоторую специальную дополнительную обработку. Пока это касается только объектов языка Python, то никаких проблем не возникает; если встречается объект, в котором реализован провайдер `IDynamicMetaObjectProvider`, и он не является объектом языка Python, то такой объект перенаправляется в класс `CompatibilityInvokeBinder`, который наследуется от класса `InvokeBinder` и который может правильно обрабатывать чужие объекты.

Если при возвращении к семантике хост-языка выполнить привязку операцию не удается, исключительное состояние не создается; вместо этого возвращается объект `DynamicMetaObject`, представляющий собой ошибку. Затем средство привязки хост-языков обработает его так, как это делается в самом хост-языке; например, доступ к отсутствующему члену объекта языка IronPython из гипотетической реализации JavaScript может вернуть неопределенное значение `undefined`, тогда как то же самое действие в объекте языка JavaScript из IronPython будет причиной возникновения ошибки `AttributeError`.

Возможность языков работать с динамическими объектами практически бесполезна без возможности языков загружать и выполнять код, написанный на других языках, а для этого в среде DLR для других языков предоставляется единый механизм.

8.9. Хост-среда

Кроме обеспечения общих особенностей реализации языков, в среде DLR также предоставлен единый используемый *хост-интерфейс* (*hosting interface*). Хост-интерфейс используется хост-языком (обычно статическим языком, например, языком C#) для выполнения кода, написанного на другом языке, например, на языке Python или Ruby. Это обычный метод, который позволяет конечным пользователям расширять приложение, а среда DLR двигается на шаг вперед, делая триадальным использование любого скриптового языка, реализация которого есть в среде DLR. В хост-интерфейсе присутствуют следующие четыре ключевые компоненты: среда времени выполнения *runtime*, движки *engines*, средства работы с исходным кодом *sources* и механизмы *scopes*, реализующие среду выполнения кода.

Среда времени выполнения `ScriptRuntime`, как правило, используется совместно всеми динамическими языками, имеющимися в приложении. Среда времени выполнения обрабатывает все текущие ссылки, которые присутствуют на загруженный языках, предоставляет методы быстрого выполнения файлов, а также предоставляет методы создания новых движков. Для простых скриптовых задач, среда времени выполнения является лишь интерфейсом, которым необходимо поль-

зоваться, но в DLR также предлагаются классы, обеспечивающие лучший контроль над тем, как выполняются скрипты.

Как правило, для каждого скриптового языка используется только один движок `ScriptEngine`. Поскольку используется протокол мета-объектов среды DLR, то это значит, что программа может загружать скрипты, написанные на различных языках, а объекты, созданные на каждом отдельном языке, могут легко взаимодействовать между собой. Движок представляет собой обертку вокруг контекста `LanguageContext`, специфического для каждого конкретного языка (например, `PythonContext` или `RubyContext`), и используется для выполнения кода, загружаемого из файла или строк, а также для выполнения операций над динамическими объектами для языков, в которых изначально среда DLR не поддерживается (например, язык C# до версии .NET 4). Движки являются поточно-ориентированными и поскольку каждый поток работает в своей собственной области видимости, параллельно могут выполнять несколько скриптов. Также предоставляются методы, применяемые при написании исходных кодов скриптов и обеспечивающие более детальный контроль над выполнением скрипта.

Фрагменты кода, который должен выполняться, содержатся внутри класса `ScriptSource`; этот компонент привязывается объект `SourceUnit`, в котором хранится фактический код, к движку `ScriptEngine`, для которого создан исходный код. Этот класс позволяет компилировать этот код (в результате создается объект `CompiledCode`, который можно кэшировать) и непосредственно его выполнять. Если фрагмент кода будет выполняться неоднократно, то его лучше сначала скомпилировать, а затем выполнять скомпилированный код; для скриптов, которые выполняются однократно, код лучше просто непосредственно выполнить.

Когда, в конце концов, потребуется выполнить код, то с помощью механизма `ScriptScope` нужно будет создать среду, в которой он будет выполняться. Эта среда используется для хранения всех переменных, используемых в скрипте, и в ней может выполнять, если это необходимо, предварительную загрузку переменных из хост-среды. Это позволяет хост-среде передавать пользовательские объекты в скрипт, когда тот начинает работать; например, когда скрипт работает, то можно с помощью редактора изображений предоставить метод для доступа к пикселям изображения. После того, как скрипт закончит работать, из среды работы скрипта можно будет прочитать переменные, которые он создал. Другой основной особенностью среды выполнения является то, что она изолирует работу скрипта, поэтому одновременно можно загружать и выполнять несколько скриптов, которые не будут мешать друг другу.

Важно отметить, что все эти классы предоставляются средой DLR, а не языком; из реализации языка в движке используется только класс контекста языка `LanguageContext`. В контексте языка предоставляются все функциональные возможности: загрузка кода, создание среды выполнения, компиляция, выполнение кода и выполнение операций над динамическими объектами — то, что необходимо хост-среде, а в среде DLR предоставляются классы, обеспечивающий более удобный интерфейс доступа к этим функциональным возможностям. Благодаря этому один и тот же код хост-среды можно использовать для любого языка, разрабатываемого в среде DLR.

Чтобы реализовать динамический язык, написанный на языке С (например, оригинальные языки Python и Ruby), в динамическом языке необходимо написать специальный код-обертку, причем это требуется повторять для каждого поддерживаемого скриптового языка. Хотя есть программы, например, SWIG, которые упрощают эту работу, добавить в программу скриптовый интерфейса языка Python или Ruby и предоставить внешним скриптам возможность работать с объектной моделью все еще продолжает оставаться нетривиальной задачей. Однако в среде .NET добавление возможности использования скриптов выполняется достаточно просто с помощью помощи настройки среды выполнения, загрузки в среду выполнения частей программы и применение метода `ScriptScope.SetVariable()` для того, чтобы объекты программы стали доступными для скриптов. Можно в течение нескольких минут добавить в приложение среды .NET поддержку выполнения скриптов, что является огромным преимуществом среды DLR.

8.10. Общая структура

Поскольку среда DLR эволюционировала из отдельной библиотеки в часть среды CLR, есть компоненты, которые находятся в среде CLR (точки вызовов, деревья выражений, средства привязки, генерация кода и динамические метаобъекты), и компоненты, которые являются частью проекта языков IronLanguages, имеющего открытый исходный код (хостинг, интерпретатор и несколько других небольших компонентов, которые здесь не рассматриваются). Компоненты, находящиеся в среде CLR, также включены в состав проекта IronLanguages в виде компонента `Microsoft.Scripting.Core`. Компоненты среды DLR разделены на две составляющие — `Microsoft.Scripting` и `Microsoft.Dynamic` — к первой относятся интерфейсы API хостинга, а во второй находится код взаимодействия с объектами СОМ, интерпретатор и некоторые другие общие компоненты динамических языков.

Сами языки также разделены на две составляющие: в `IronPython.dll` и `IronRuby.dll` реализованы сами языки (синтаксические анализаторы, средства привязки и т.д.), а в `IronPython.Modules.dll` и `IronRuby.Libraries.dll` реализованы те части стандартной библиотеки, которые в классических реализациях языков Python и Ruby написаны на языке C.

8.11. Усвоенные уроки

Среда DLR является полезным примером языково-нейтральной платформы для динамических языков, создаваемых поверх статический среды времени выполнения. Технологии, требующиеся для достижение высокой производительности динамического кода, достаточно нетривиальны при их надлежащей реализации, поэтому они были реализованы в среде DLR и были сделаны доступными для каждой реализации динамического языка.

`IronPython` и `IronRuby` являются хорошими примерами того, как создавать язык поверх среды DLR. Реализации очень похожи друг на друга, т.к. они разрабатывались в одно и то же время тесно сотрудничающими командами, но в реализации есть существенные различия. Совместная разработка нескольких разных языков (`IronPython`, `IronRuby`, прототип `JavaScript` и таинственный `VBx` — полностью динамическая версия VB), а также динамических возможностей языков C# и VB позволили во время разработки достаточно хорошо протестировать проект DLR.

Реальная разработка `IronPython`, `IronRuby` и DLR выполнялась совсем не так, как это было в то время в большинстве проектов Microsoft — с первого дня использовалась интерактивная модель agile-разработки с непрерывной интеграцией. Это позволило вносить изменения очень быстро, сразу, как это становилось необходимым, что было очень хорошо, поскольку уже на ранних стадиях разработки среда DLR стала разрабатываться в связке с динамическими возможностями языка C#. Хотя тесты среды DLR проходили очень быстро, всего лишь десятки секунд или около того, запуск тестов языка занимал слишком много времени (комплект тестов `IronPython` выполнялся в течение приблизительно 45 минут даже при использовании распараллеливания); улучшение этой ситуации смогло бы повысить скорость итераций. В конечном счете, эти итерации сходились к текущей разработке DLR, что, если это рассматривать по отдельности, выглядит чрезвычайно сложно, но когда это собиралось вместе, то, в общем, выглядело достаточно хорошо.

Разработка среды DLR в связке с языком C# была исключительно важным решением, поскольку это убедило, что у DLR есть сфера применения и «целевое назначение», но когда динамические возможности языка C# были реализованы, изменился политический климат (совпадающий с экономическим спадом) и с стороны компании была прекращена поддержка языков Iron. Например, интерфейс API хостинга никогда не входил в состав фреймворка .NET (и маловероятно, что это когда-нибудь произойдет); что означает, что, хотя все объекты все еще могут взаимодействовать так, как описано выше, в PowerShell 3, который также основан на среде DLR, используется набор интерфейсов API хостинга, абсолютно отличающийся от используемых в `IronPython` и `IronRuby`.

Некоторые из членов команды, разрабатывающей DLR, продолжали работать над библиотекой «компилятор как сервис» языка C#, имеющей кодовое название «Roslyn» и поразительно похожей на интерфейс API хостинга IronPython и IronRuby. Но, благодаря чуду лицензирования открытого исходного кода, языки Iron продолжают выживать и даже процветают.

9. ITK

9.1. Что такое ITK?

[ITK, the Insight Toolkit](#) является библиотекой для анализа изображений, которая была разработана по инициативе и при практической поддержке [Национальной библиотеки медицины США](#). ITK может рассматриваться в качестве полезной энциклопедии алгоритмов для анализа изображений, в частности алгоритмов обработки изображений с помощью фильтров, сегментации и геометрической коррекции изображений. Библиотека была разработана консорциумом, включающим университеты, коммерческие компании и индивидуальных разработчиков со всего мира. Разработка ITK началась в 1999 году и после недавнего десятилетия код библиотеки был подвергнут рефакторингу, направленному на удаление устаревшего кода и внесение улучшений, позволяющих использовать библиотеку в течение следующих десяти лет.

9.2. Возможности архитектуры

Программные тулкиты обычно очень тесно связаны со своими сообществами разработчиков. Они подстраиваются друг под друга в продолжающемся последовательном цикле. Программное обеспечение постоянно модифицируется до того момента, пока оно начинает удовлетворять требованиям сообщества, при этом сообщество само адаптируется к возможностям тулкита, устанавливая действия, которые программный компонент позволяет или запрещает выполнять. Поэтому для лучшего понимания особенностей архитектуры ITK очень полезно иметь представление о том, с какими проблемами постоянно сталкивалось сообщество разработчиков и как эти проблемы обычно решались.

Характер зверя

Если вы не понимаете характер зверей, знания в области их анатомии вам не очень помогут.
- Dee Hock, *One from Many: Visa and the Rise of Chaordic Organization*

При рассмотрении типичной задачи по анализу изображений исследователь или инженер возьмет исходное изображение, улучшит некоторые характеристики этого изображения, скажем, убрав шумы и повысив контрастность, после чего перейдет к установлению некоторых параметров изображения, таких, как углы и острые края. Этот тип обработки изначально хорошо совместим с архитектурой конвейера данных, как показано на [Рисунке 9.1](#).

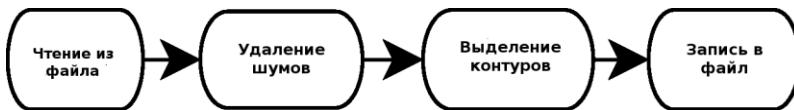


Рисунок 9.1: Конвейер обработки изображения

Для иллюстрации этого утверждения на [Рисунке 9.2](#) показан снимок головного мозга, полученный в ходе магнитно-томографического резонансного исследования (magnetic resonance image (MRI)) и результат его обработки с помощью медианного фильтра для снижения уровня шума, а также результат применения фильтра выделения контуров для идентификации границ анатомических структур.

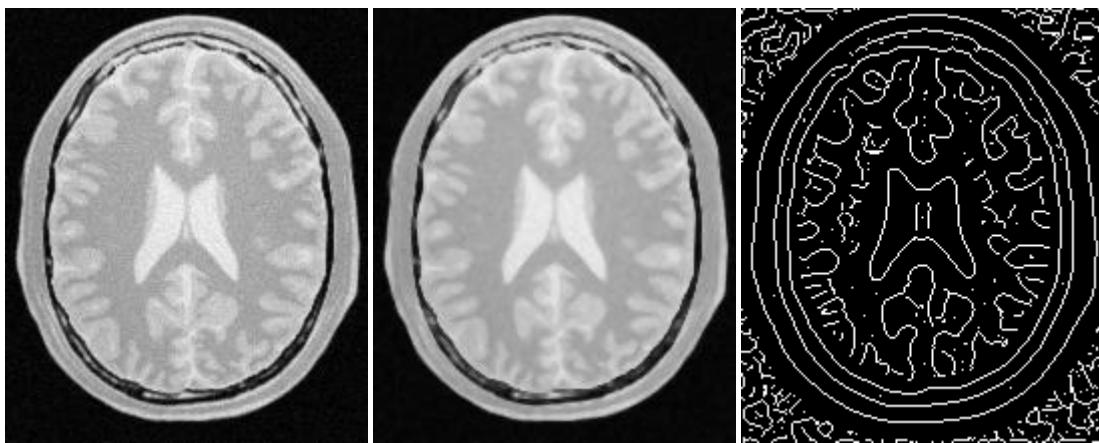


Рисунок 9.2: Снимок головного мозга в ходе МРТ, результат использования медианного фильтра, результат использования фильтра выделения контуров

Для решения каждой из этих задач сообщество разработчиков алгоритмов анализа изображений реализовало множество алгоритмов и продолжает вести работу над новыми. Вы можете спросить: "Почему они продолжают делать это?" и ответ на ваш вопрос будет заключаться в том, что в процессе обработки изображений используется комбинация научных знаний, инженерных расчетов, искусства и навыков "подготовки" материала. Утверждение о том, что существует комбинация алгоритмов, которая позволяет "корректно" решить задачу анализа изображения настолько же ошибочно, насколько и утверждение о том, что существует "правильный" тип шоколадного десерта для обеда. Вместо доведения алгоритмов до совершенства сообщество разработчиков пытается создать набор разнообразных инструментов, в случае использования которого вы можете быть уверены в том, что при решении задачи по анализу изображения, набор параметров анализа не окажется недостаточным. Конечно же, такие возможности имеют свою цену. Ценой в данном случае являются сложности выбора из множества различных инструментов тех, которые необходимы, причем инструменты могут использоваться совместно в различных комбинациях для получения похожих результатов.

Сообщество разработчиков алгоритмов анализа изображений тесно связано с сообществом исследователей. Нередко оказывается, что определенные группы исследователей связаны с семействами алгоритмов, которые они разработали. Эта традиция "брэндинга" и в некоторой степени "маркетинга" приводит к ситуации, в которой лучшим вариантом, который программный тулkit может предложить сообществу, является предложение полного набора реализаций алгоритмов, которые оно может попробовать, а также подобрать для создания рецепта, удовлетворяющего его потребностям.

Существует несколько причин, по которым тулkit ИТК был спроектирован и реализован в виде обширной коллекции частично независимых, но связанных инструментов, являющихся фильтрами изображений (*image filters*), многие из которых могут использоваться для решения аналогичных задач. В этом контексте присутствует некоторый уровень "избыточности" - например, предоставление трех различных реализаций фильтра Гаусса рассматривается не как проблема, а как полезная возможность, так как различные реализации могут быть взаимозаменяемыми для работы в различных условиях и повышения эффективности в зависимости от размера изображения, количества процессоров и размера ядра метода Гаусса, который может быть задан используемым приложением обработки изображений.

Также тулkit создавался в виде ресурса, который должен развиваться и постоянно обновлять самого себя по мере появления новых и улучшения реализаций существующих алгоритмов, замены существующих алгоритмов, а также разработки новых инструментов для удовлетворения срочных требований, предъявляемых новыми технологиями создания медицинских изображений.

Вооружившись знаниями, полученными в ходе краткого обзора ежедневных дел специалиста по анализу изображений из сообщества разработчиков ITK, мы можем перейти к более подробному рассмотрению наиболее важных особенностей архитектуры:

- Модульность
- Конвейер обработки данных
- Фабрики
- Фабрики ввода/вывода
- Поточная передача данных
- Повторное использование кода
- Возможности сопровождения кода

Модульность

Модульность является одной из основных характеристик ITK. Это требование, которое вытекает из метода работы членов сообщества специалистов по анализу изображений при решении поставленных перед ними задач. Большинство задач по анализу изображений подразумевает обработку одного или нескольких изображений с помощью комбинации фильтров для улучшения качества или выделения определенной информации из изображения. Следовательно, не существует одного большого объекта для обработки изображений, а вместо него используется несметное количество малых объектов. Из этого структурного характера задачи обработки изображений логически вытекает необходимость реализации программного обеспечения в виде большой коллекции фильтров для обработки изображений, которые могут комбинироваться различными способами.

Также в данном случае некоторые фильтры для обработки изображений объединяются в семейства, внутри которых некоторые из особенностей их реализации могут отличаться. Это позволяет произвести изначальное разделение фильтров для обработки изображений на модули и группы модулей.

Следовательно, модульность реализуется на трех основных уровнях в рамках ITK:

- Уровень фильтра
- Уровень семейства фильтров
- Уровень группы семейств фильтров

На уровне фильтра обработки изображений ITK содержит около 700 фильтров. Учитывая то, что фреймворк ITK реализован с использованием языка программирования C++, это уровень, на котором каждый из этих фильтров реализован с помощью класса C++ в соответствии с шаблонами объектно-ориентированного проектирования. На уровне семейства фильтров ITK группирует фильтры в соответствии с характером обработки изображения, которую они выполняют. Например, все фильтры, выполняющие преобразования Фурье, будут объединены в рамках модуля. На уровне языка C++ модули ставятся в соответствие директориям в дереве исходного кода и библиотекам после компиляции исходного кода в бинарный формат. ITK содержит около 120 таких модулей. Каждый модуль содержит:

1. Исходный код фильтров изображений, принадлежащих рассматриваемому семейству.
2. Набор файлов конфигурации, которые описывают метод сборки модуля и содержат список зависимостей между рассматриваемым и сторонними модулями.
3. Набор модульных тестов, соответствующих каждому из фильтров.

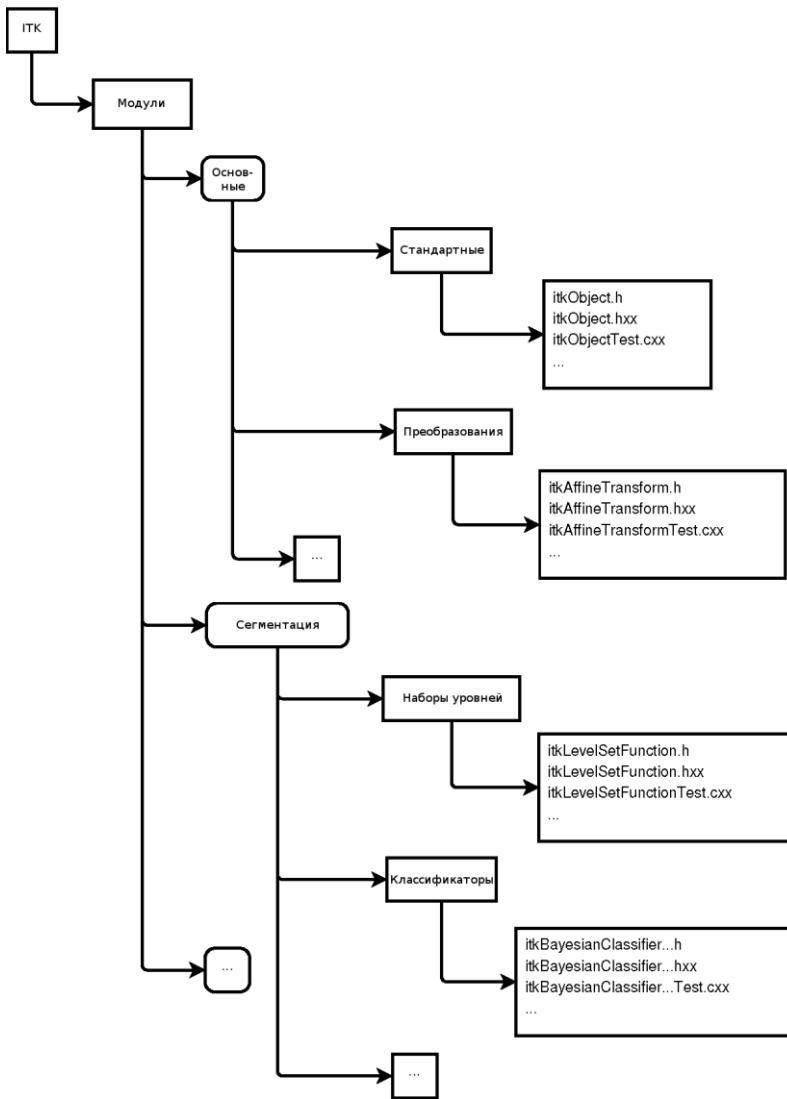


Рисунок 9.3: Иерархическая структура групп, модулей и классов

На уровне групп производится в большей степени концептуальное деление, не зависящее от характеристик программного обеспечения и упрощающее поиск фильтров в дереве исходного кода. Группы ассоциированы с такими высокоуровневыми концепциями, как фильтрация (Filtering), сегментация (Segmentation), геометрическая коррекция (Registration) и ввод/вывод (IO). Иерархическая структура показана на Рисунке 9.3. В данный момент в комплекте поставки ITK насчитывается 124 модуля, которые в свою очередь распределены между 13 основными группами. Модули значительно различаются в размерах. Распределение размеров модулей в байтах отображено на Рисунке 9.4.

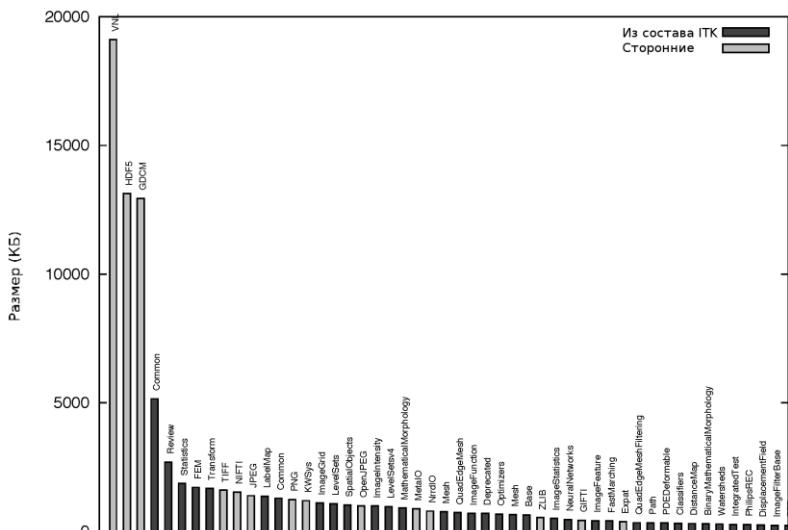


Рисунок 9.4: Распределение размеров 50 наиболее больших модулей ИТК в КБ

Применяющийся в ИТК модульный подход также справедлив и в случае сторонних библиотек, которые не являются частью тулкита, но от которых зависит его работа и которые распространяются в комплекте поставки тулкита вместе с остальным кодом для удобства пользователей. Отдельными примерами этих сторонних библиотек являются библиотеки для работы с файлами изображений различных форматов: HDF5, PNG, TIFF, JPEG и OpenJPEG, а также другие. Вопрос о сторонних библиотеках поднимается по той причине, что на их счет приходится около 56 процентов размера комплекта поставки ИТК. Это обстоятельство отражает обычной подход, применяемый при сборке приложений с открытым исходным кодом для существующих платформ. Распределение размеров сторонних библиотек не обязательно соответствует архитектурным особенностям ИТК, так как мы используем эти полезные библиотеки в том виде, в каком они были разработаны. Однако, сторонний код перераспределяется вместе с кодом тулкита и его отделение являлось одной из ключевых директив процесса формирования модулей.

Распределение размеров модулей было приведено выше, так как оно является мерой качества разделения кода на модули. Можно заметить, что модульность кода представлена в виде непрерывного диапазона от экстремума, соответствующего расположению всего кода в одном модуле в виде монолитной версии до экстремума разделения всего кода на очень большой набор модулей одинакового размера. Распределение размеров было инструментом, используемым для отслеживания степени выполнения процесса разделения кода на модули, особенно для уверенности в том, что не было оставлено больших блоков кода в одном и том же модуле в случае, когда не имеется реальных логических зависимостей для такой группировки.

Модульная архитектура ИТК позволяет выполнить и упрощает:

- Сокращение количества и четкое выявление зависимостей между модулями
- Принятие кода от сообщества разработчиков
- Определение качественных параметров модуля (например, степени покрытия кода)
- Сборку выбранных частей тулкита
- Упаковку выбранных частей тулкита для повторного распространения
- Поддержание постоянного развития тулкита путем добавления новых модулей

Процесс разделения кода на модули позволил четко идентифицировать и описать зависимости между различными частями тулкита, так как они были перенесены в модули. Во многих случаях эта деятельность позволила выявить искусственные или некорректные зависимости, которые были заданы в рамках тулкита в течение длительного промежутка времени и не были замечены тогда, когда большая часть кода находилась в файлах из нескольких групп большого размера.

Польза от получения качественных параметров двояка. Во-первых, появляется возможность сделать разработчиков ответственными за модули, развитие которых они поддерживают. Во-вторых, появляется возможность занятия инициативами по очистке кода, в ходе которых несколько разработчиков в течение короткого времени работают над повышением качества определенного модуля. При работе с небольшими частями тулита проще оценивать эффективность своей работы и поддерживать интерес и мотивацию разработчиков.

В качестве повторения отменим, что структура тулита отражает организацию сообщества разработчиков, а также в некоторых случаях процессы, которые были введены для постоянного развития и контроля качества программного обеспечения.

Конвейер данных

Ступенчатый характер задач анализа изображений изначально обуславливает выбор архитектуры конвейера данных для основной инфраструктуры обработки данных. Конвейер данных позволяет выполнять следующие действия:

- *Объединение фильтров:* Фильтры изображений из набора могут быть объединены один с другим, составляя цепочку обработки, в рамках которой над исходными изображениями производится последовательность операций.
- *Исследование эффекта от изменения параметров:* Как только фильтры объединяются в цепочку, становится достаточно просто изменить параметры любого фильтра цепочки и исследовать эффект, оказываемый на результирующее изображение благодаря изменению этих параметров.
- *Потоковая передача данных из памяти:* Изображения большого размера могут быть обработаны путем модификации исключительно блоков изображения в каждый момент времени. Таким образом становится возможной обработка изображений большого размера, которые не могут быть размещены в оперативной памяти каким-либо образом.

На Рисунках 9.1 и 9.2 уже было продемонстрировано упрощенное представление конвейера данных с точки зрения обработки изображений. Фильтры изображений обычно имеют числовые параметры, которые используются для управления поведением фильтра. В любой момент, когда значение одного из числовых параметров фильтра изменяется, конвейер данных устанавливает отметку "устаревшее" для результирующего изображения и располагает информацией о том, что этот определенный фильтр и все фильтры после него, использовавшие его вывод, должны быть повторно применены к изображению. Эта возможность конвейера данных упрощает исследование эффекта от изменения параметров фильтров при использовании минимального объема вычислительных мощностей для проведения каждого эксперимента.

Процесс обновления данных конвейера может быть реализован таким образом, что в каждый момент времени будут обрабатываться только части изображений. Это необходимый механизм для поддержки функции потоковой передачи данных. На практике этот процесс контролируется путем внутренней передачи спецификации `RequestRegion` от следующего в цепочке фильтра к используемому. Это взаимодействие осуществляется с помощью внутреннего API и не раскрывается для разработчиков приложений.

Для рассмотрения более конкретного примера, предположим, что если фильтр размытия изображения Гаусса ожидает изображение разрешением 100x100 пикселей на входе, которое должно быть сформировано медианным фильтром изображения, то фильтр размытия может запросить у медианного фильтра создание только четверти изображения, которая будет представлена фрагментом размером 100x25 пикселей. Этот запрос впоследствии может быть передан дальше по цепочке с учетом того, что каждый промежуточный фильтр может добавить дополнительную границу к размеру фрагмента изображения для предоставления запрошенного выходного размера этого фрагмента. Потоковая передача данных будет рассмотрена более подробно [ниже](#).

И изменение параметров заданного фильтра, и изменение определенного фрагмента изображения путем обработки с помощью фильтра приведут к установлению отметки "устаревшее" на результирующее изображение, указывающей на необходимость повторного использования заданного и всех следующих за ним в цепочке фильтров из состава конвейера данных.

Объекты процесса и данных

Для сохранения базовой структуры конвейера были спроектированы два основных типа объектов. Это `DataObject` и `ProcessObject`. Объект `DataObject` является абстракцией над классами для хранения данных; например, изображений и геометрических сеток. Объект `ProcessObject` является абстракцией над фильтрами изображений и фильтрами сеток, с помощью которых обрабатываются данные. Объекты `ProcessObject` принимают объекты `DataObject` в качестве исходных данных и проводят с ними какие-либо алгоритмические трансформации, подобные проиллюстрированным на Рисунке 9.2.

Объекты `DataObject` генерируются объектами `ProcessObject`. Эта цепочка обычно начинается с чтения данных для объекта `DataObject` с диска, например, с помощью объекта `ImageFileReader`, являющегося одним из типов объекта `ProcessObject`. Объект `ProcessObject`, создавший заданный объект `DataObject`, является единственным объектом, который может модифицировать данный объект `DataObject`. Этот результирующий объект `DataObject` объединен с входом другого объекта `ProcessObject`, находящегося далее в цепочке.

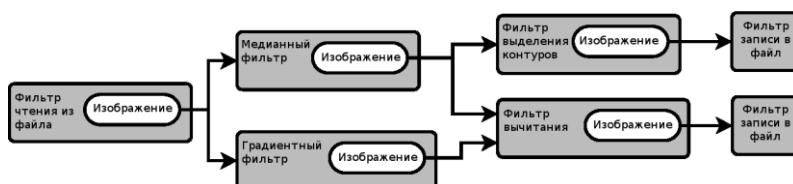


Рисунок 9.5: Взаимодействия между объектами `ProcessObject` и `DataObject`

Эта последовательность проиллюстрирована на Рисунке 9.5. Один и тот же объект `DataObject` может быть передан на вход множества объектов `ProcessObject`, как показано на рисунке, где объект `DataObject` создается путем чтения из файла в начале конвейера. В данном случае объект для чтения из файла является экземпляром класса `ImageFileReader`, а объект `DataObject`, создаваемый в результате чтения - экземпляром класса `Image`. Для некоторых фильтров также необходимо требовать по два объекта `DataObject` в качестве входных данных, как в случае фильтра вычитания, показанного в правой части того же рисунка.

Объединение объектов `ProcessObject` и `DataObject` является побочным эффектом процесса формирования конвейера. С точки зрения разработчика приложений элементы конвейера связаны друг с другом путем использования последовательности вызовов, затрагивающих объекты `ProcessObject` и аналогичных следующим:

```

writer->SetInput ( canny->GetOutput() );
canny->SetInput ( median->GetOutput() );
median->SetInput ( reader->GetOutput() );
  
```

Однако, во внутреннем представлении результатом этих вызовов является не соединение объекта `ProcessObject` со следующим объектом `ProcessObject`, а объединение предыдущего объекта `ProcessObject` с объектом `DataObject`, созданным следующим объектом `ProcessObject`.

Внутренняя структура цепочки конвейера формируется с помощью трех типов объединений:

- Объект `ProcessObject` хранит список указателей на результирующие объекты `DataObject`. Результирующие объекты `DataObject` находятся в распоряжении и контролируются объектом `ProcessObject`, с помощью которого они были созданы.
- Объект `ProcessObject` хранит список указателей на объекты `DataObject`, содержащие исходные данные. Объекты `DataObject`, содержащие исходные данные, находятся в распоряжении и контролируются предыдущим объектом `ProcessObject`.

- Объект `DataObject` хранит указатель на создавший его объект `ProcessObject`. Следовательно, объект `ProcessObject` также владеет и контролирует объект `DataObject`.

Этот набор внутренних связей используется позднее для выполнения запросов к последующим и предыдущим фильтрам в рамках конвейера. В ходе всех этих взаимодействий объект `ProcessObject` удерживает контроль и распоряжается объектом `DataObject`, который он создал. Последующие фильтры получают доступ к информации о заданном объекте `DataObject` с помощью ссылок в форме указателей, которые создаются в результате вызовов методов `SetInput()` и `GetOutput()` без получения контроля над входными данными. В практических целях фильтры должны рассматривать входные данные как объекты, предназначенные только для чтения. Такое поведение устанавливается на уровне API с помощью ключевого слова `const` языка C++ для аргументов методов `SetInput()`. Как правило, в рамках ITK создается корректный в отношении констант внешний API, даже с учетом того, что во внутреннем представлении константы иногда переназначаются некоторыми операциями конвейера.

Иерархия классов конвейера

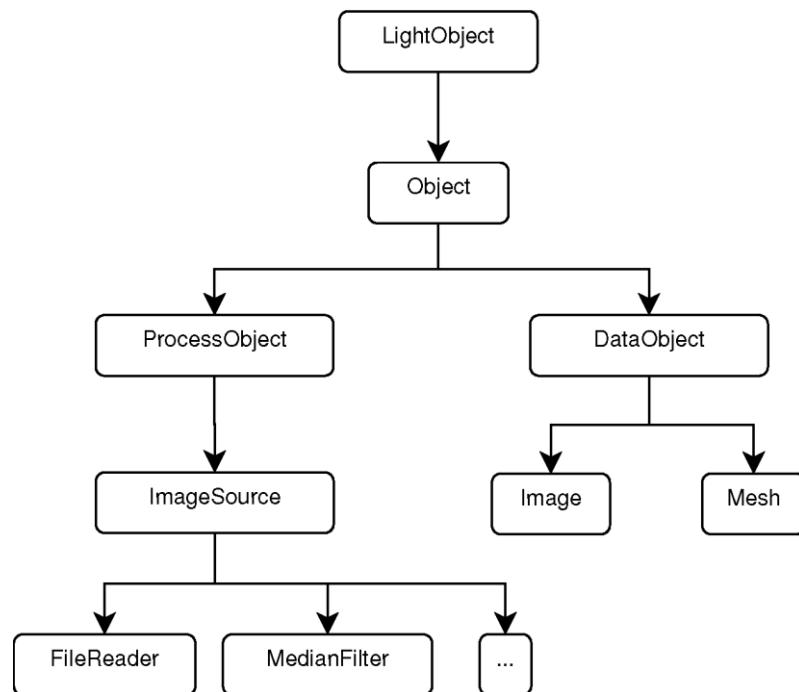


Рисунок 9.6: Иерархия классов `ProcessObject` и `DataObject`

Начальное проектирование и реализация конвейера данных фреймворка ITK происходили по аналогии с Visualization Toolkit (VTK), развитым проектом на момент начала разработки ITK. (Обратитесь к тому 1 книги *"Архитектура приложений с открытым исходным кодом"*).

На Рисунке 9.6 показана иерархия объектов конвейера ITK. В частности следует отметить взаимосвязь между основными объектами `Object`, `ProcessObject`, `DataObject`, некоторыми классами из семейства фильтров и семейства хранилищ данных. В рамках данной абстракции любой объект, который предназначен для передачи фильтру или создается в ходе вывода данных фильтром, должен наследоваться от класса `DataObject`. Все фильтры, которые выводят и принимают данные должны наследоваться от класса `ProcessObject`. Механизмы обмена информацией о данных, требуемые для передачи данных по конвейеру, реализованы частично в рамках класса `ProcessObject` и частично в рамках класса `DataObject`.

Классы `LightObject` и `Object` являются иллюстрацией дихотомии классов `ProcessObject` и `DataObject`. Классы `LightObject` и `Object` предоставляют такие стандартные функции, как API для обмена объектами `Events` и средства поддержки многопоточности.

Внутренние процессы конвейера

На Рисунке 9.7 представлена UML-диаграмма последовательности вызовов, описывающая взаимодействия между объектами `ProcessObject` и `DataObject` в рамках упрощенного конвейера, состоящего из объектов `ImageFileReader`, `MedianImageFilter` и `ImageFileWriter`.

Совокупность взаимодействий делится на четыре фазы:

- Обновление информации о выходных данных (обратная последовательность вызовов)
- Обновление запрошенного фрагмента изображения (обратная последовательность вызовов)
- Обновление выходных данных (обратная последовательность вызовов)
- Генерация данных (прямая последовательность вызовов)

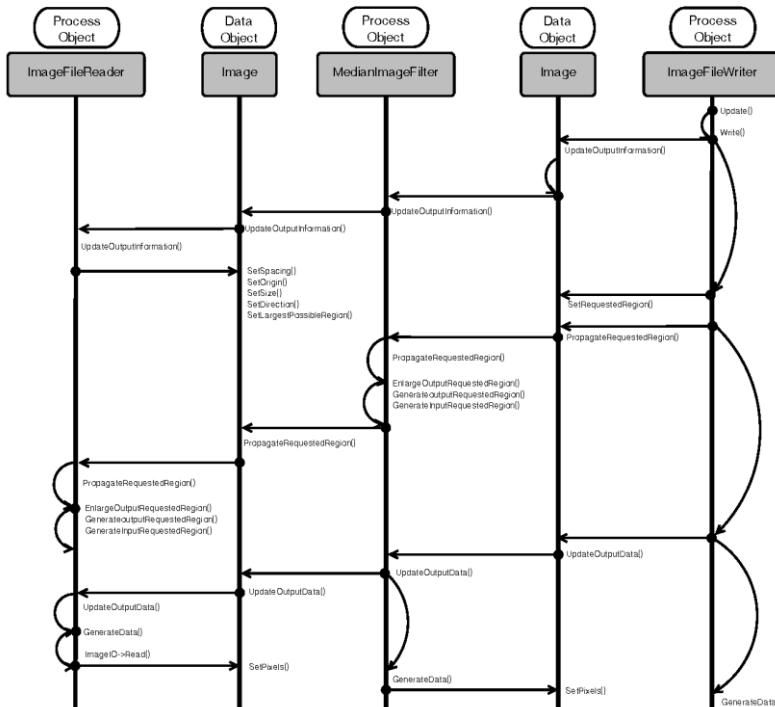


Рисунок 9.7: UML-диаграмма последовательности вызовов

Процесс работы конвейера начинается с вызова приложением метода `Update()` последнего фильтра конвейера; в этом конкретном примере этот фильтр представлен объектом `ImageFileWriter`. Вызов метода `Update()` инициирует первую обратную последовательность вызовов. Это последовательность, которая начинается с последнего фильтра конвейера и распространяется в направлении первого фильтра.

Целью первой фазы является получение ответа на вопрос "Как много данных будет сгенерировано?" В форме кода этот вопрос реализован в рамках метода `UpdateOutputInformation()`. С помощью этого метода каждый фильтр рассчитывает объем данных изображения, который может быть получен на выходе в случае обработки заданного исходного объема данных. Учитывая то, что объем исходных данных должен быть известен до того, как фильтр сможет дать ответ на вопрос об объеме выходных данных, вопрос должен переадресовываться предыдущим фильтрам до того момента, как будет достигнут исходный фильтр, который может самостоятельно ответить на вопрос. В этом конкретном примере исходный фильтр представлен объектом `ImageFileReader`. Этот фильтр может установить объем выходных данных, получив информацию из файла изображения, который был выбран для чтения. После того, как первый фильтр конвейера отвечает на поставленный вопрос, последующие фильтры один за другим получают возможность вычисления соответствующих объемов выходных данных до того момента, как последний фильтр конвейера вычислит этот объем.

Вторая фаза, в ходе которой вызовы также осуществляются против направления конвейера, предназначена для информирования фильтров об объеме выходных данных, который они должны сгенерировать в ходе работы конвейера. Концепция запрашиваемых фрагментов (*Requested Region*) необходима для поддержки потоковой обработки данных в ИТК. Она позволяет сообщать фильтрам конвейера о том, что не следует генерировать полное изображение, а следует обрабатывать только его фрагмент, являющийся запрашиваемым фрагментом. Это очень полезно в том случае, если имеющееся изображение не может быть полностью помещено в доступную оперативную память системы. Вызовы направлены от последнего фильтра к первому, причем каждый промежуточный фильтр модифицирует размер запрашиваемого фрагмента с учетом необходимых дополнительных границ исходного изображения, которые могут понадобиться фильтру для генерации фрагмента изображения заданного размера. В этом конкретном примере медианный фильтр обычно добавляет границу толщиной в 2 пикселя к исходному изображению. Таким образом, если фильтр записи запрашивает фрагмент размером 500 x 500 пикселей у медианного фильтра, медианный фильтр в свою очередь запросит фрагмент размером 502 x 502 пикселя у фильтра чтения, так как медианному фильтру с настройками по умолчанию требуется фрагмент размером 3 x 3 пикселя для вычисления значения одного пикселя результирующего изображения. Эта фаза реализована в форме кода в рамках метода `PropagateRequestedRegion()`.

Третья фаза направлена на запуск выполнения расчетов в отношении данных запрошенного фрагмента. В ходе этой фазы вызовы также осуществляются против направления конвейера, а в форме кода она реализована в рамках метода `UpdateOutputData()`. Так как каждому фильтру требуются исходные данные для обработки и формирования выходных данных, вызовы изначально передаются предшествующим соответствующим фильтрам, что подразумевает распространение вызовов против направления конвейера. После возврата данных предыдущим фильтром, текущий фильтр приступает непосредственно к их обработке.

Четвертая и финальная фаза предусматривает выполнение вызовов по направлению конвейера и заключается в непосредственной обработке данных каждым из фильтров. В форме кода эта фаза представлена в рамках метода `GenerateData()`. Направление, совпадающее с направлением конвейера, является последствием не того, что каждый фильтр осуществляет вызовы методов следующего фильтра, а того факта, что вызовы `UpdateOutputData()` осуществляются в последовательности от первого к последнему фильтру конвейера. Таким образом, все вызовы осуществляются в направлении конвейера в соответствии со временем вызовов, а не в соответствии с тем, какой фильтр осуществляет запросы. Это пояснение важно, так как конвейер ИТК соответствует концепции *Pull Pipeline*, в которой данные запрашиваются с конца конвейера и управление логикой осуществляется также с конца конвейера.

Фабрики

Одним из фундаментальных требований, предъявляемых во время проектирования к ИТК, является возможность поддержки множества платформ. Это требование основывается на желании максимально расширить распространение тулкита путем реализации возможности его использования сообществом вне зависимости от предпочтительных платформ участников. В рамках проекта ИТК был реализован шаблон проектирования фабрики (*Factory design pattern*) для решения задачи по поддержке фундаментальных различий множества аппаратных и программных платформ без ущерба совместимости решения с каждой из платформ.

Шаблон проектирования фабрик в ИТК использует имена классов в качестве ключей для реестра конструкторов классов. Регистрация фабрик происходит в процессе работы приложения и может быть осуществлена путем простого размещения динамических библиотек в соответствующих директориях, где приложения на основе ИТК производят их поиск при запуск. Эта возможность позволяет использовать существующий механизм для реализации модульной архитектуры очевидным и прозрачным образом. В результате упрощается процесс разработки расширяемых приложе-

ний для анализа изображений, удовлетворяющих требованиям предоставления постоянно расширяющегося набора функций анализа изображений.

Фабрики ввода/вывода

Механизм фабрик особенно важен при осуществлении операций ввода/вывода.

Обработка различных систем с помощью фасадов

Сообщество, занимающееся исследованием изображений, разработало очень большой набор форматов файлов для сохранения данных изображений. Многие из этих форматов проектировались и разрабатывались для специфических нужд и, следовательно, хорошо оптимизированы для хранения специфических типов изображений. В результате в сообществе постоянно разрабатываются и рекомендуются для использования новые форматы файлов изображений. Предсказывая эту ситуацию, команда разработчиков ITK спроектировала подходящую для простого расширения функций архитектуру ввода/вывода, в рамках которой достаточно просто регулярно добавлять поддержку все большего и большего набора форматов файлов.

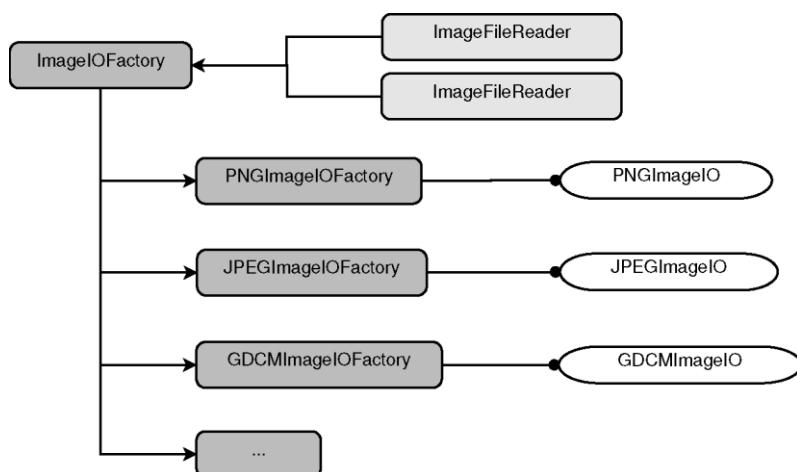


Рисунок 9.8: Зависимости фабрик ввода/вывода

Эта расширяемая архитектура для осуществления операций ввода/вывода основана на механизме фабрик, описанном в предыдущем разделе. Основное отличие заключается в том, что в случае системы для операций ввода/вывода фабрики ввода/вывода регистрируются в специализированном реестре, управляемом базовым классом `ImageIOFactory`, изображенным в верхнем левом углу Рисунка 9.8. Сами функции чтения и записи данных для различных форматов файлов изображений реализованы в рамках семейства классов `ImageIO`, изображенного справа на Рисунке 9.8. Эти служебные классы предназначены для создания экземпляров по требованию в тот момент, когда пользователь осуществляет чтение или запись изображения. Эти служебные классы не раскрываются в рамках кода приложений. Вместо прямого обращения к этим классам приложения должны взаимодействовать с классами фасадов:

- `ImageFileReader`
- `ImageFileWriter`

Это два класса, при использовании которых в приложении может применяться подобный код:

```
reader->SetFileName("../image1.png");
reader->Update();
```

или:

```
writer->SetFileName("../image2.jpg");
writer->Update();
```

В обоих случаях вызов метода `Update()` приводит к запуску конвейера, с которым соединены рассматриваемые объекты `ProcessObject`. И объект чтения, и объект записи данных ведут себя как дополнительный фильтр конвейера. В случае объекта для чтения данных вызов метода `Update()` приводит к чтению соответствующего файла изображения и размещению данных в памяти. В случае объекта записи вызов метода `Update()` приводит к выполнению операций конвейера, в результате чего объект записи получает входные данные, и, наконец, завершается записью изображения на диск в файл определенного формата.

Эти классы фасадов скрывают от разработчика приложений внутренние сложности, возникающие из-за особенностей каждого из форматов файлов. Они скрывают даже признаки самого существования формата. Эти фасады спроектированы таким способом, что большую часть времени разработчикам приложений не требуется знать о том, какие форматы файлов могут быть прочитаны приложением. Стандартное приложение может просто использовать подобный код:

```
std::string filename = this->GetFileNameFromGUI();
writer->SetFileName( filename );
writer->Update();
```

Эти вызовы будут работать аналогично в том случае, если значение переменной `filename` будет представлено одной из следующих строк:

- image1.png
- image1.jpeg
- image1.tiff
- image1.dcm
- image1.mha
- image1.nii
- image1.nii.gz

причем расширения файлов указывают на различные форматы файлов в каждом случае.

Знайте тип пикселя

Несмотря на содействие со стороны фасадных классов чтения и записи, разработчику приложений следует заботиться о том, какой тип пикселей требуется обрабатывать приложению. В контексте работы с медицинскими изображениями разумно ожидать того, что разработчик приложения будет знать о том, содержит ли исходное изображение магнитно-резонансную томограмму, маммографию или компьютерную томограмму и, следовательно, позаботится о выборе подходящего типа пикселя и разрешения изображения для каждого из этих различных типов изображений. Эти особенности типов изображений могут доставлять неудобства в случае использования настроек приложения, при которых пользователи хотят иметь возможность чтения *любого* типа изображения, что часто встречается при быстром создании прототипов и обучении. В контексте развертывания приложений для работы с медицинскими изображениями для эксплуатации в клиниках, однако, ожидается, что тип пикселей и разрешение изображений будут четко заданы и смогут определяться на основе типа предназначенного для обработки изображения. Конкретный пример, в котором приложение работает с трехмерными магнитно-резонансными томограммами, выглядит следующим образом:

```
typedef itk::Image< signed short, 3 > MRImageType;
typedef itk::ImageFileWriter< MRImageType > MRIWriterType;
MRIWriterType::Pointer writer = MRIWriterType::New();
```

```
writer->Update();
```

Однако, существует ограничение того, насколько особенности форматов файлов изображений могут быть скрыты от разработчика приложений. Например, при чтении изображений из файлов форматов DICOM или RAW разработчику придется использовать дополнительные вызовы для точного указания характеристик имеющегося формата. Файлы формата DICOM наиболее часто встречаются в медицинских учреждениях, а файлы формата RAW все еще являются необходимым злом, предназначенным для обмена данными в процессе исследований.

Объединенные, но разделенные

Автономный характер каждой фабрики ввода/вывода и служебный класс ImageIO также были затронуты процессом разделения на модули. Обычно класс ImageIO зависит от специализированной библиотеки, предназначеннной для работы со специфическим форматом файлов. Такими форматами, например, являются PNG, JPEG, TIFF и DICOM. В этих случаях сторонняя библиотека рассматривается как независимый модуль и специализированный код класса ImageIO, являющийся интерфейсом между кодом ITK и кодом сторонней библиотеки, также размещается в модуле. Таким образом, специфические приложения могут ограничить использование множества форматов файлов, которые не входят в сферу их применения и работать только с теми форматами файлов, которые окажутся полезными в ожидаемых сценариях применения данного приложения.

Как и в случае со стандартными фабриками, загрузка фабрик ввода/вывода может быть осуществлена в процессе работы приложения из динамических библиотек. Этот гибкий процесс загрузки фабрик упрощает использование специализированных самостоятельно разработанных форматов файлов без необходимости включения поддержки таких форматов файлов непосредственно в состав тулкита ITK. Загружаемые фабрики ввода/вывода были одним из наиболее успешных архитектурных решений проекта ITK. Они позволили достаточно просто разрешить сложную ситуацию без усложнения или запутывания кода. Не так давно подобная архитектура ввода/вывода была реализована для управления процессом чтения и записи файлов, содержащих пространственные преобразования в рамках семейства классов Transform.

Потоковая передача данных

Изначально тулкит ITK разрабатывался как набор инструментов для обработки изображений, полученных проектом [Visible Human Project](#). В то время было совершенно ясно, что такой большой набор данных не сможет быть размещен в оперативной памяти компьютеров, которые обычно были доступны участникам сообщества исследователей медицинских изображений. Также данный набор данных все еще не может быть размещен в памяти стандартных настольных компьютеров, использующихся на сегодняшний день. Следовательно, одним из требований к разработке проекта Insight Toolkit было обеспечение возможности потоковой передачи данных изображения по конвейеру данных. Точнее, возможности обработки изображений больших размеров путем передачи блоков изображения через конвейер данных и последующей сборки результирующих блоков в конце конвейера.

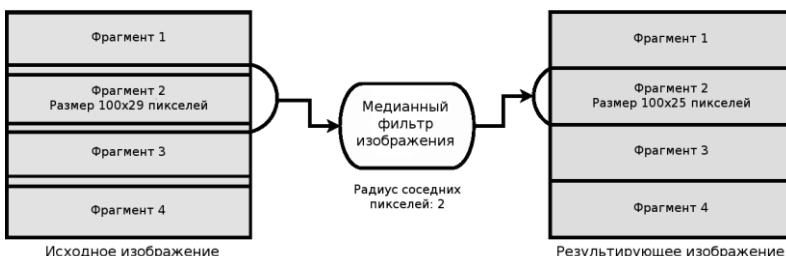


Рисунок 9.9: Иллюстрация процесса потоковой передачи изображения

Метод разделения области изображения проиллюстрирован на Рисунке 9.9 для конкретного примера медианного фильтра. Медианный фильтр вычисляет значение результирующего пикселя как статистическую медиану значений соседних пикселей исходного изображения. Размер границы из соседних пикселей является числовым параметром фильтра. В данном случае мы установили значение, равное 2 пикселям и это означает, что мы будем использовать соседние пиксели в радиусе 2 пикселей вокруг нашего результирующего пикселя. В результате нам необходим фрагмент изображения размером 5x5 пикселей с результирующим пикселем в центре и квадратной границей в 2 пикселя вокруг него. Обычно это называется Манхэттен-радиусом (Manhattan radius). В тот момент, когда медианный фильтр получает запрос расчета значений пикселей запрашиваемого фрагмента выходного изображения, он обращается к предыдущему фильтру и просит его предоставить фрагмент большего, чем заданный с помощью спецификации запрашиваемого фрагмента размера, причем размер увеличивается на количество пикселей границы, в нашем случае на 2 пикселя. В специфическом случае, представленном на Рисунке 9.9, при запросе фрагмента 2 размером 100x25 пикселей, медианный фильтр передает запрос предыдущему фильтру, причем размер фрагмента устанавливается равным 100x29 пикселей. Вертикальный размер фрагмента, равный 29 пикселям, вычисляется как сумма 25 пикселей и размеров двух границ по 2 пикселя каждой. Следует отметить, что горизонтальный размер фрагмента не увеличивается в данном случае, так как этот размер является максимальным для рассматриваемого исходного изображения; следовательно, увеличенный размер, равный 104 пикселям (сумма 100 пикселей и размеров двух границ по 2 пикселя) уменьшен в соответствии с максимальным размером изображения, которое равно 100 пикселям по горизонтали.

Фильтры ITK, работающие с соседними пикселями, обрабатывают граничные условия одним из трех стандартных способов: устанавливают нулевое значение пикселей вне изображения, зеркально отражают значения пикселей по границе или повторяют значения граничных пикселей вне изображения. В случае медианного фильтра используется граничное условие Неймана нулевого потока (zero-flux Neumann boundary condition), которое просто означает, что значения пикселей вне границы фрагмента изображения устанавливаются равными значениям последних найденных пикселей в границах изображения.

Хорошо хранимым литературой по обработке изображений небольшим секретом является тот факт, что большинство сложностей реутилизации фильтров изображений относится к корректной обработке граничных условий. Это определенный симптом различий между теоретическими заданиями, которые встречаются во многих книгах, и практическим опытом разработки программного обеспечения для обработки изображений. В рамках проекта ITK эта задача была решена путем реализации набора классов итераторов изображения и соответствующего семейства классов для расчета граничных условий. Два этих семейства вспомогательных классов скрывают от фильтров изображений сложности управления граничными условиями в N-ном количестве плоскостей.

Процесс потоковой передачи данных изображения начинается вне фильтра, обычно с помощью классов `ImageFileWriter` или `StreamingImageFilter`. Два этих класса реализуют функции потоковой передачи данных, разделяя изображение на несколько фрагментов в соответствии с требованиями разработчика. После этого в ходе вызова их метода `Update()` они выполняют итерацию, запрашивая каждый из промежуточных фрагментов изображения. На этом этапе используются возможности API `SetRequestRegion()`, проиллюстрированного с помощью Рисунка 9.7. Этот вызов позволяет ограничить область расчета значений пикселей изображения с помощью фильтров фрагментом этого изображения.

Код приложения, позволяющий осуществить потоковую обработку данных, выглядит аналогично следующему:

```
median->SetInput( reader->GetOutput() );
median->SetNeighborhoodRadius( 2 );
writer->SetInput( median->GetOutput() );
```

```
writer->SetFileName( filename );
writer->SetNumberOfStreamDivisions( 4 );
writer->Update();
```

единственным новым элементом которого является вызов `SetNumberOfStreamDivisions()`, который устанавливает количество фрагментов, на которые будет разделено изображение для потоковой обработки с помощью конвейера. Для соответствия Рисунку 9.9 мы использовали значение, равное четырем и устанавливающее количество фрагментов для разделения изображения. Это значит, что объект `writer` выполнит запуск фильтра `median` четыре раза, причем каждый раз фильтру будет передаваться отличная структура запрашиваемого фрагмента.

Существуют интересные сходства между процессом потоковой обработки данных и процессом параллельной работы нескольких экземпляров заданного фильтра. В обоих случаях процесс основывается на возможности разделения задачи по обработке изображения путем разделения изображения на отдельные фрагменты, которые могут обрабатываться независимо. В случае потоковой обработки данных фрагменты изображения обрабатываются последовательно один за другим, а в случае параллельной обработки фрагменты изображения привязываются к отдельным потокам, которые в свою очередь привязываются к отдельным ядрам процессора. В конечном счете алгоритмический характер работы фильтров устанавливает то, возможно ли разделить результирующее изображение на фрагменты, которые будут обрабатываться отдельно на основании соответствующего набора фрагментов исходного изображения. В ITK функции потоковой и параллельной обработки практически ортогональны в том смысле, что существует API для управления процессом потоковой обработки существует отдельный API, предназначенный для поддержки реализации базовых функций параллельных вычислений с использованием множества потоков и фрагментов разделяемой памяти.

Потоковая обработка, к сожалению, не совместима со всеми типами алгоритмов. Особые случаи, в которых потоковая обработка не может быть осуществлена:

- Итерационные алгоритмы, в которых для расчета значения пикселя в каждой итерации в качестве исходных данных требуются значения соседних пикселей. Это справедливо для большинства алгоритмов решения уравнений в частных производных, таких, как алгоритмы анизотропной диффузии, деформационной коррекции `demons` и обработки с использованием структур частых наборов уровней.
- Алгоритмы, которые требуют наличия полного набора пикселей исходного изображения для расчета значения одного из пикселей результирующего изображения. Фильтры на основе преобразований Фурье и бесконечной импульсной характеристики (IIR), такие, как рекурсивный фильтр Гаусса являются примерами фильтров этого класса.
- Алгоритмы распространения фрагментов или слоев, в ходе работы которых модификация пикселей также происходит с помощью итерации, но расположение фрагментов и слоев не позволяет предсказуемо разделить изображение на блоки. Алгоритмы сегментации изображений, обработки их с использованием структур редких наборов уровней, некоторые реализации математических морфологических операций и некоторые формы алгоритмов "watershed" являются примерами этого типа алгоритмов.
- Алгоритмы коррекции изображений в том случае, если им требуется доступ ко всем данным исходного изображения для расчета метрических значений в каждой итерации их циклов оптимизации.

К счастью, с другой стороны, структура конвейера данных ITK поддерживает потоковую передачу данных для различных фильтров преобразований, используя тот факт, что все фильтры создают свое результирующее изображение и, следовательно, не перезаписывают область памяти, содержащую исходное изображение. Это приводит к затратам памяти, так как конвейеру приходится одновременно резервировать память и для исходных, и для результирующих изображений.

Фильтры, осуществляющие такие операции, как переворот изображений, перестановка осей и геометрические изменения, попадают в эту категорию. В этих случаях конвейер данных управляет сопоставлением фрагментов исходного и результирующего изображений, требуя, чтобы каждый фильтр предоставлял метод с названием `GenerateInputRequestedRegion()`, который принимает в качестве аргумента прямоугольную область результирующего изображения. Этот метод производит расчет значений пикселей прямоугольного фрагмента исходного изображения, которые потребуются этому фильтру для расчета значений пикселей этого определенного прямоугольного фрагмента результирующего изображения. Этот постоянный обмен данными в рамках конвейера данных позволяет поставить в соответствие каждому блоку результирующего изображения соответ-

ствующий блок исходного изображения, который требуется для проведения расчета значений пикселей.

Если быть более точным, то можно сделать вывод о том, что ITK поддерживает поточную передачу данных изображения, но только при использовании "поточных" алгоритмов. Тем не менее, для того, чтобы быть прогрессивными в отношении оставшихся алгоритмов, мы должны трактовать это утверждение не как жалобу о том, что "невозможно реализовать поточную обработку данных при использовании этих алгоритмов", а как утверждение о том, что "наш стандартный подход к потоковой передаче данных не совместим с этими алгоритмами" на данный момент и мы надеемся, что в будущем сообщество изобретет новые техники для решения этой проблемы.

9.3. Выученные уроки

Повторное использование

Принцип повторного использования может быть также назван "уменьшением избыточности". В случае ITK это достигается с помощью подхода, включающего три принципа:

- Во-первых, применение объектно-ориентированного программирования и в особенности корректной процедуры создания иерархий классов, в которых стандартные функции реализуются в базовых классах.
- Во-вторых, использование парадигмы обобщенного программирования, реализуемое путем повсеместного использования шаблонов языка C++, а также реализации функций программы, идентифицируемых как шаблоны.
- В-третьих, широкое использование макросов C++ также позволило повторно использовать стандартные фрагменты кода, которые требуются в миллиарде мест тулкита.

Большая часть этих пунктов сегодня может показаться банальной и очевидной, но в момент начала разработки ITK в 1999 году некоторые из них не были настолько очевидными. В частности, при поддержке большинством компиляторов языка C++ шаблонов, эта поддержка в точности не соответствовала однозначному стандарту. Даже сегодня такие решения, как применение парадигмы обобщенного программирования и использование реализации с широким применением шаблонов продолжают вызывать споры в сообществе. Это утверждение подчеркивается фактом существования сообществ, участники которых предпочитают использовать ITK с помощью уровня кода для совместимости с языками Python, Tcl или Java.

Обобщенное программирование

Применение парадигмы обобщенного программирования было одной из определяющих особенностей реализации ITK черт. В 1999 году это решение было сложным, так как в то время поддержка компиляторами шаблонов C++ была значительно фрагментированной и стандартная библиотека шаблонов (Standard Template Library (STL)) все еще рассматривалась в некоторой степени как экзотическое дополнение.

Обобщенное программирование было применено в ITK путем использования шаблонов C++ для обобщенной реализации концепций и повышения степени повторного использования кода таким способом. Стандартным примером использования параметризации шаблонов C++ в ITK является класс `Image`, экземпляр которого может быть создан следующим образом:

```
typedef unsigned char PixelType;
const unsigned int Dimension = 3;
typedef itk::Image< PixelType, Dimension > ImageType;
ImageType::Pointer image = ImageType::New();
```

В этом коде разработчик приложения выбирает тип, используемый для представления пикселей изображения, а также количество плоскостей изображения в виде сетки в пространстве. В примере

мы выбрали для использования 8-битные пиксели, представленные с помощью значений типа `unsigned char` для 3-х мерного изображения. Благодаря низкоуровневой обобщенной реализации, в ITK возможно создание экземпляров классов изображений для любых типов пикселей с любым количеством плоскостей.

Для возможности записи этих выражений разработчикам ITK пришлось реализовать класс `Image` с особой осторожностью в отношении предположений о типе пикселей. Как только разработчик приложения создал экземпляр класса изображения определенного типа, он может создавать объекты этого типа или перейти к созданию экземпляров классов фильтров изображений, типы которых, в свою очередь, зависят от типа изображения. Например: `typedef itk::MedianImageFilter<ImageType, ImageType> FilterType; FilterType::Pointer median = FilterType::New();`

Алгоритмические особенности некоторых фильтров изображений ограничивают набор актуальных типов пикселей, который они поддерживают. Например, некоторые фильтры изображений ожидают, что тип пикселей изображения будет представлен целочисленными скалярными величинами, в то время, как некоторые другие фильтры ожидают типа пикселя в виде векторов из величин с плавающей точкой. При создании экземпляров классов изображений с неподходящими типами пикселей эти фильтры станут причиной ошибок компиляции или ошибочных результатов расчетов значений пикселей изображения. Для предотвращения некорректного создания объектов и упрощения поиска причин ошибок компиляции в рамках ITK был применен метод концептуальных проверок (*concept checking*), основанный на принудительном использовании определенных ожидаемых возможностей типов с целью раннего выявления ошибок с выводом понятных человеку сообщений об ошибках.

Шаблоны C++ также используются в определенных системах тулкита в форме шаблонного метапрограммирования (Template Metaprogramming) для повышения производительности кода, а в особенности для разворачивания циклов, которые используются для расчета значений низкоразмерных векторов и матриц. По иронии судьбы, со временем мы обнаружили, что определенные компиляторы научились определять места, где необходимо осуществить разворачивание циклов и в некоторых случаях им больше не требуется выражений шаблонного метапрограммирования.

Знать, когда остановиться

Также существует риск того, что при разработке будет "сделано очень много хороших вещей", подразумевающий риск чрезмерного использования шаблонов или чрезмерного использования макросов. Достаточно просто зайти слишком далеко и в итоге создать новый языка программирования на основе C++, реализованный с помощью шаблонов и макросов. Это тонкая грань, которая требует постоянного внимания со стороны команды разработчиков для уверенности в том, что возможности языка программирования используются должным образом без злоупотреблений.

В качестве конкретного примера можно привести широко используемую схему явного именования типов с помощью оператора `typedef` C++, которая оказалась достаточно важной. Эта практика позволяет выполнить две задачи: с одной стороны, она позволяет использовать понятные для человека и информативные имена, описывающие характер и назначение типа; с другой стороны, она позволяет быть уверенным в том, что тип используется последовательно в рамках тулкита. В качестве примера следует упомянуть о том, что в ходе рефакторинга тулкита при подготовке версии 4.0 были приложены большие усилия для того, чтобы установить случаи использования таких целочисленных типов C++, как `int`, `unsigned int`, `long` и `unsigned long` и заменить их на типы, названные в соответствии с надлежащими концепциями, связанными с представляемыми переменными данными. Эта часть работы, направленной на предоставление возможности использования 64-битных типов для обработки изображений объемом более четырех гигабайт на всех платформах, была наиболее сложной. Данная задача имела крайне важное значение для последующего использования ITK в области микроскопии и дистанционного зондирования, где изображения объемом в десятки гигабайт встречаются достаточно часто.

Возможность поддержки кода

Архитектура удовлетворяет требованиям, направленным на снижение сложности поддержки кода.

- Модульность (на уровне классов)
- Большое количество файлов небольших размеров
- Повторное использование кода
- Повторное использование шаблонов

Эти характеристики сокращают затраты труда на сопровождение кода в следующих случаях:

- Модульность (на уровне классов) позволяет применять техники разработки через тестирование на уровне фильтров изображения или в общем на уровне классов ИТК. Строгая дисциплина тестирования, применяемая к небольшим и модульным фрагментам кода, позволяет воспользоваться полезной возможностью сокращения количества фрагментов кода, в которых могут быть скрыты ошибки, а также естественным результатом разделения кода на модули является упрощение поиска и устранения дефектов.
- Большое количество файлов небольших размеров позволяет упростить выделение фрагментов кода для работы над ними определенных разработчиков и упростить отслеживание дефектов в том случае, если они ассоциированы с определенными коммитами в системе контроля версий. Дисциплина работы с файлами небольшого размера также ведет к применению золотого правила функций и классов: выполнять одну задачу и выполнять ее правильно.
- Повторное использование кода: При повторном использовании кода (вместо копирования и вставки или повторной реализации) качество кода увеличивается ввиду повышения уровня его исследования, что происходит из-за его применения в различных обстоятельствах. Это обстоятельство позволяет рассмотреть код большим количеством глаз или, по крайней мере, получить эффект и, соответственно, преимущества, описанные законом Линуса (Linus's Law): "При достаточном количестве глаз все ошибки лежат на поверхности".
- Повторное использование шаблонов упрощает работу мэйнтайнеров, трудозатраты которых на самом деле составляют более 75% трудозатрат, вложенных в разработку проекта в течение всего периода его существования. Использование шаблонов проектирования, которые постоянно повторяются в различных местах кода, значительно упрощает понимание разработчиком того, что код делает или должен делать сразу же после открытия файла.

По мере того, как разработчики вовлекаются в процесс регулярной поддержки кода, они могут столкнуться с некоторыми "стандартными ошибками", а именно:

- Предположения о том, что некоторые фильтры принимают определенные типы пикселей входных и выходных изображений, но не устанавливают их с помощью проверок типов и концептуальных проверок, а также эти типы не описаны в документации.
- Разработка кода, не являющегося удобным для чтения. Это одна из наиболее часто встречающихся сложностей в рамках любого программного обеспечения, новые реализации алгоритмов которого появляются в сообществе исследователей. Часто разработчики реализуют код, который "просто работает" и забывают о том, что задачей кода является не только исполнение в ходе работы приложения, но и возможность его простого чтения следующим разработчиком. Стандартные хорошие правила разработки "чистого кода" - например, разработка небольших функций, которые выполняют одну задачу и только одну задачу (принцип единой ответственности (Single Responsibility Principle) и принцип наименьшей неожиданности (Principle of Least Surprise)) наряду с корректным именованием переменных и функций - обычно игнорируются тогда, когда исследователи вдохновлены реализацией своего нового великолепного алгоритма.
- Игнорирование случаев неудачного завершения задач и обработки ошибок. Стандартной практикой является приоритетное рассмотрение "счастливых случаев" в процессе обработки данных и отсутствие кода для обработки всех случаев, в которых процесс работы приложений может протекать некорректно. Пользователи тулита довольно быстро сталкиваются с такими случаями сразу же после того, как они начинают разработку и развертывание приложений в реальных условиях.
- Недостаточное тестирование. Требуется большое количество дисциплинарных мероприятий для того, чтобы следовать практике разработки через тестирование, особенно при приоритетной разработке тестов и реализации функций только после того, как они будут протестированы. Практически всегда ошибки в коде встречаются в тех местах, которые были пропущены при реализации кода тестирования.

Благодаря практикам взаимодействия сообществ разработчиков приложений с открытым исходным кодом, многие из этих ошибок в конце концов обнаруживаются с помощью вопросов, обычно задаваемых в списках рассылки или в отправляемых напрямую пользователями сообщениях об ошибках. После рассмотрения множества подобных случаев разработчики учатся писать код, который "подходит для сопровождения". Некоторые из свойств этого кода относятся и к стилю оформления кода, и к самой организации кода. На наш взгляд, разработчик достигает мастерства только после некоторого времени работы - хотя бы года - по сопровождению кода и рассмотрения "всех случаев, в которых код может работать некорректно".

Невидимая рука

Программное обеспечение должно выглядеть так, как будто оно было разработано одним человеком. Лучшими являются разработчики, создающие код, который может развиваться любым другим разработчиком, если основного разработчика сбьет автобус, упоминаемый в известной фразе. Мы достигли осознания того, что любой признак "индивидуального стиля разработчика" является указателем на дефект программного обеспечения.

Для применения и распространения информации о необходимости использования единого стиля оформления кода могут использоваться следующие подтвердившие свою высокую эффективность инструменты:

- [KWStyle](#) используется для автоматической проверки соответствия кода определенному стилю оформления. Этот инструмент является упрощенной системой разбора кода на языке C++, проверяющей его соответствие стилю оформления и сообщающей о любых несоответствиях этому стилю.
- [Germit](#) используется для регулярных обзоров кода. Этот набор инструментов может использоваться для двух целей: с одной стороны он преграждает путь в кодовую базу для недостаточно проработанного кода, выявляя содержащиеся в нем ошибки, неточности и дефекты в ходе итерационных циклов проверки, в рамках которых другие разработчики улучшают рассматриваемый код. С другой стороны он позволяет создать виртуальный тренировочный лагерь, в котором новые разработчики будут учиться у опытных разработчиков (термин "опытные" следует понимать как *совершившие все ошибки и знающие обо всех секретах...*) методам повышения качества кода и обхода известных проблем, которые были выявлены в ходе циклов поддержки программного обеспечения.
- Точки вызова программ Git, позволяющие использовать инструменты KWStyle и Germit, а также позволяющие выполнить некоторые внутренние проверки. Например, проект ITK использует точки вызова программ Git, с помощью которых предотвращается добавление в репозиторий кода с символами табуляции или с завершающими строку пробелами.
- Команда разработчиков также исследовала возможность использования инструмента [Uncrustify](#) для приведение кода к единому стилю.

Следует подчеркнуть, что единый стиль кода не просто улучшает его эстетическую привлекательность, а на самом деле оказывает влияние на экономические показатели проекта. Исследование совокупной стоимости владения (Total Cost of Ownership (TCO)) программными проектами установило, что в течение цикла жизни проекта стоимость его поддержки будет составлять около 75% от TCO и, учитывая то, что стоимость обслуживания регулярно повышается, она обычно превышает затраты на полную разработку проекта за пять первых лет цикла жизни программного проекта. (Обратитесь к книге "*Software Development Cost Estimating Handbook*", Volume I, Naval Center for Cost Analysis, Air Force Cost Analysis Agency, 2008.) Затраты на сопровождение проекта оцениваются в 80% от той работы, которую на самом деле выполняют разработчики программного обеспечения и при осуществлении этой деятельности большая часть времени разработчиков отводится на чтение кода, созданного другим человеком и выяснение того, для выполнения каких действий этот код предназначен. (обратитесь к книге "*Clean Code, A Handbook of Agile Software Craftsmanship*", Robert C. Martin, Prentice Hall, 2009). Единый стиль кода чудесным образом сокращает время, затрачиваемое разработчиками на погружение в код из нового файла проекта и понимание этого кода до того, как они смогут как-либо модифицировать его. К тому же, сокращается вероятность того, что разработчики неправильно поймут код и модифицируют его, создав новые ошибки в ходе добросовестной попытки исправления старых (*The Art of Readable Code*, Dustin Boswell, Trevor Foucher, O'Reilly, 2012).

Ключевым фактором эффективного применения этих инструментов является проверка соответствия следующим критериям:

- Доступность для всех разработчиков, из-за чего мы предпочитаем инструменты с открытым исходным кодом.
- Использование на регулярной основе. В случае проекта ITK эти инструменты были интегрированы в систему ночных и последовательных сборок, формируемых с помощью системы [CDash](#).
- Использование как можно ближе к тому месту, где был разработан новый код для немедленного исправления отклонений от стиля оформления, таким образом разработчики будут быстро узнавать о том, какие методы разработки нарушают стиль оформления кода.

Рефакторинг

Проект ITK был начат в 2000 году и постоянно развивался до 2010 года. В 2011 году благодаря финансовым вливаниям из федеральных инвестиций команда разработчиков получила по истине

уникальную возможность занятия рефакторингом кода. Финансирование было осуществлено Национальной библиотекой медицины в рамках этапа реализации инициативы по оздоровлению экономики Америки и reinvestированию (American Recovery and Reinvestment Act (ARRA)). Эта работа не была второстепенной. Представьте, что вы работали над частью программного обеспечения в течение более чем десяти лет и вам предоставили возможность улучшения ее кода; что бы вы изменили?

Такая возможность реализации расширенного рефакторинга появляется очень редко. В течение предыдущих десяти лет мы ежедневно проводили небольшие локальные рефакторинги, очищая определенные уголки тулкита после того, как оказывались в них. Этот продолжительный процесс очистки и улучшения кода использовался благодаря преимуществам взаимодействия участников сообществ проектов с открытым исходным кодом, а безопасность процесса обеспечивалась инфраструктурой тестирования на основе CDash, которая обычно проверяет около 84% кода тулкита. Следует отметить, что в отличие от нашего показателя, среднее покрытие кода системами промышленного тестирования оценивается только в 50%.

Среди множества вещей, измененных в процессе рефакторинга, наиболее относящимися к вопросам архитектуры являются:

- Начало процесса разделения кода на модули
- Стандартизация целочисленных типов
- Исправления в области применения операторов `typedef` для возможности работы с изображениями, объем которых превышает 4 GB на всех платформах
- Пересмотр процесса разработки программного обеспечения:
 - Миграция с CVS на Git
 - Начало выполнения обзоров кода с использованием Gerrit
 - Начало выполнения тестов по запросу с использованием CDash@home
 - Улучшение метода загрузки данных для модульного тестирования
- Окончание поддержки устаревших компиляторов
- Улучшение поддержки ввода/вывода для различных форматов файлов изображений, среди которых:
 - DICOM
 - JPEG2000
 - TIFF (BigTIFF)
 - HDF5
- Реализация фреймворка для поддержки расчетов средствами видеoadаптера
- Реализация функций для поддержки обработки видео
 - Добавление кода поддержки технологии OpenCV
 - Добавление кода поддержки технологии VXL

Поддержка развития проекта, основанная на инкрементальных модификациях - производящаяся в ходе выполнения таких задач, как добавление возможностей фильтра изображения, улучшение производительности заданного алгоритма, работа на основе сообщения об ошибке и улучшение документации определенных фильтров изображений - хорошо функционирует при локальном улучшении определенных классов C++. Однако, расширенный рефакторинг необходим для инфраструктурных модификаций, которые затрагивают большое количество классов по всем направлениям, таких, как те, что были перечислены ранее. Например, набор изменений, необходимых для поддержки файлов объемом более 4 GB был, вероятно, одним из самых больших патчей, которые когда-либо применялись по отношению к ITK. Этот процесс потребовал модификации сотен классов и не мог быть выполнен инкрементально без серьезных неудобств. Процесс разделения кода на модули является еще одним примером задачи, которая не могла быть выполнена инкрементально. На самом деле этот процесс повлиял на всю организацию тулкита, на то, как работает инфраструктура тестирования, как производится управление тестовыми данными, как тулкит упаковывается и распространяется, а также на то, как новые фрагменты исходного кода должны быть сформированы для включения в состав тулкита в будущем.

Воспроизводимость

Одним из ранних уроков, выученныхных при разработке ИТК, являлся тот факт, что многие опубликованные описания алгоритмов из области интересов команды разработчиков были не настолько просты в реализации, как нам казалось. Исследователи, специализирующиеся на вычислительных алгоритмах, зачастую чрезмерно радуются разработке нового алгоритма и избегают практической работы по созданию программного обеспечения, заявляя о том, что это "всего лишь детали реализации".

Такое безответственное отношение достаточно опасно для всей области исследований, так как оно подразумевает отрицание необходимости личного опыта, связанного с реализацией кода и его корректным использованием. В результате большинство опубликованных описаний алгоритмов просто не воспроизводимо и когда исследователи и студенты пытаются использовать подобные техники, они затрачивают большое количество времени в процессе реализации и предоставляют вариации исходной работы. На самом деле на практике достаточно сложно проверить, совпадает ли реализация с тем, что описано на бумаге.

Проект ИТК нарушил такое положение вещей в хорошем смысле и восстановил культуру самостоятельной работы в этой области исследований, где уже привыкли к теоретическим обоснованиям и научились избегать работы по проведению экспериментов. Новая культура, привнесенная проектом ИТК, является практической и прагматической культурой, в рамках которой о преимуществах программного обеспечения судят по практическим результатам его использования, а не по видимой сложности, которая считается преимуществом в некоторых научных публикациях. Оказывается, что на практике наиболее эффективными методами обработки изображений являются те методы, которые выглядели бы очень просто для включения в научную публикацию.

Культура создания воспроизводимых алгоритмов является продолжением философии разработки через тестирование и систематически приводит к повышению качества программного обеспечения; большей ясности кода, упрощению его чтения, уменьшению количества ошибок и точности реализации алгоритмов.

Для заполнения бреши, заключающейся в недостатке воспроизводимых публикаций, сообщество ИТК создало сайт [Insight Journal](#). Доступ к размещаемым на нем публикациям открыт для всех, причем от авторов публикаций требуется предоставление исходного кода, данных, параметров и тестов для возможности проверки воспроизводимости. Статьи публикуются в сети менее чем через 24 часа после отправки. После этого они становятся доступны для обзора любым участником сообщества. Читатели получают полный доступ ко всем материалам, прилагающимся к статье, а именно: исходному коду, данным, параметрам и сценариям для тестирования. *Insight Journal* стал продуктивной средой для обмена новыми фрагментами кода, которые начинают свой путь по направлению к кодовой базе проекта. *Insight Journal* недавно получил 500-ю статью и продолжает использоваться как официальный источник нового кода для добавления в состав ИТК.

10. GNU Mailman

Глава 10 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

[GNU Mailman](#) является свободным программным обеспечением для управления списками рассылки. Практически каждый, кто разрабатывает или использует свободное программное обеспечение, сталкивался со списком рассылки. Списки рассылки могут использоваться для дискуссий или объявлений, а также одновременно для двух этих целей в различной мере. Иногда списки рассылки связаны с группами новостей в Usenet или такими подобными Usenet сервисами, как [Gmane](#). Списки рассылки обычно используют архивы, содержащие историю всех отправленных в них сообщений.

Система GNU Mailman используется с начала 1990 года, когда John Viega разработал ее первую версию для объединения фанатов недавно организованной группы Dave Matthews Band, члены которой были друзьями в колледже. Эта первая версия привлекла внимание сообщества разработчиков языка Python в середине 90 годов, когда центр вселенной Python переместился из научно-исследовательского института Нидерландов [CWI](#) в национальную корпорацию исследовательских инициатив [CNRI](#) в городе Reston, штат Virginia, США. В CNRI для управления различными связанными с языком Python списками рассылок мы использовали систему Majordomo, которая была разработана с использованием языка программирования Perl. Конечно же, работающим с языком Python людям было неудобно поддерживать такой большой объем кода Perl. А еще более важным является то, что мы столкнулись со значительными сложностями при модификации приложения Majordomo для наших целей (например, при добавлении простейших возможностей для борьбы со спамом), обусловленными выбранной архитектурой приложения.

Ken Manheimer играл ведущую роль в разработке начальных версий системы GNU Mailman, при этом множество замечательных разработчиков сделало свой вклад в разработку с того момента. На сегодняшний день Mark Sapiro поддерживает стабильную ветку 2.1 в то время, как автор данной главы Barry Warsaw работает над новой версией 3.0.

Многие из оригинальных архитектурных решений, принятых John Viega, дожили в форме кода до версии 3 системы Mailman и все еще могут быть изучены при исследовании кода стабильной версии. В следующих разделах я опишу некоторые из наиболее проблемных архитектурных решений в рамках версий 1 и 2 системы Mailman, а также наши подходы к решению данных проблем в версии 3.

В начале периода эксплуатации версии 1 системы Mailman мы испытывали множество проблем, заключающихся в потере сообщений или в ошибках, приводящих к бесконечной доставке одних и тех же сообщений. Это обстоятельство подтолкнуло нас к четкой формулировке двух основных принципов, являющихся критическими для будущего успешного развития системы Mailman:

- Ни одно из сообщений не должно быть потеряно.
- Ни одно из сообщений не должно быть доставлено более чем единожды.

В версии 2 системы Mailman мы повторно спроектировали систему обработки сообщений для того, чтобы быть уверенными в том, что эти два принципа будут всегда иметь наивысший приоритет. Эта часть системы находилась в стабильном состоянии как минимум в течение последних десяти лет, что является одним из ключевых факторов повсеместной эксплуатации системы Mailman, свидетелями которой мы являемся на сегодняшний день. Несмотря на модернизацию этой подсистемы в версии 3 системы Mailman, ее архитектура и реализация остаются в большей степени неизменными.

10.1. Анатомия сообщения

Одной из основных структур данных системы Mailman является сообщение электронной почты (*email message*), представленное объектом *message*. Многие интерфейсы, функции и методы в рамках системы принимают три аргумента: объект списка рассылки, объект сообщения и словарь макетов, используемый для записи и обмена данными состояния в ходе обработки сообщения системой.

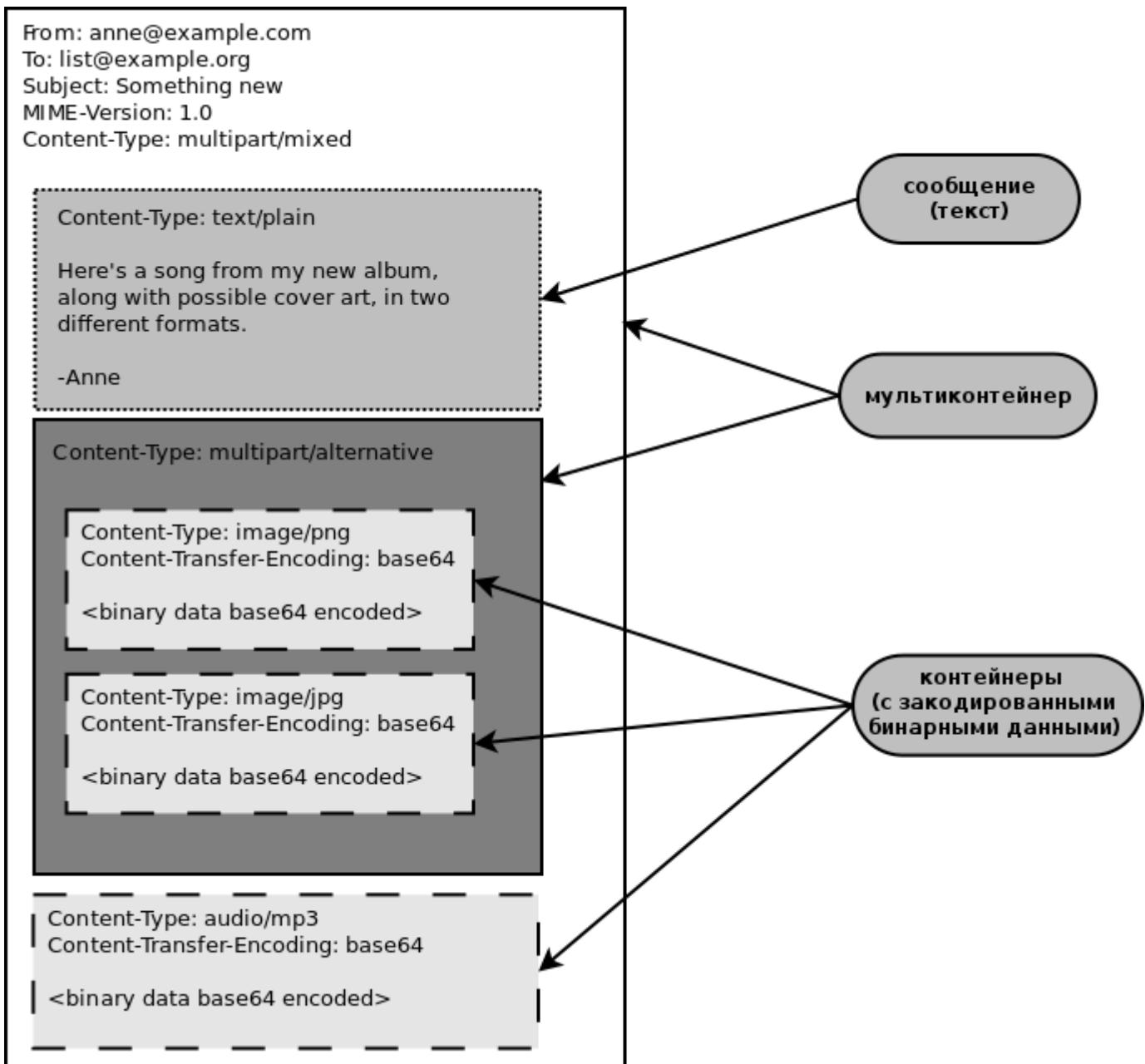


Рисунок 10.1: Сообщение, содержащее текст, изображения и звуковой файл; тип MIME

При подробном рассмотрении сообщение электронной почты оказывается простым объектом. Оно состоит из множества разделенных точкой с запятой пар ключ-значение, называемых заголовками, после которых следует пустая строка, отделяющая заголовки от сообщения. Это текстовое представление должно облегчать разбор, генерацию, исследование и манипуляции с сообщениями, но на самом деле оно стремительно усложнилось. Существует несчетное количество стандартов RFC, которые описывают все вероятные возможности, такие, как обработка сложных типов данных, представленных изображениями, аудиофайлами и другими типами. Сообщение электронной почты может содержать текст на английском языке в кодировке ASCII или текст на любом другом языке в любой существующей кодировке. Основная структура сообщения электронной почты заимствовалась снова и снова для использования в рамках других протоколов, таких, как NNTP и HTTP, причем все эти протоколы отличаются в значительной степени. В ходе нашей работы над системой Mailman была начата разработка нескольких библиотек исключительно для обработки всех возможных случаев использования данного формата (обычно называемого "RFC822" в соответствии с принятым в 1982 году [стандартом IETF](#)). Библиотеки для работы с сообщениями электронной почты, изначально разрабатываемые для применения в рамках системы GNU Mailman, были включены в стандартную библиотеку языка Python, где их разработка продолжилась в направлении улучшения стабильности работы и соответствия стандартам.

Сообщения электронной почты могут выступать в качестве контейнеров для других типов данных, как это описано в различных стандартах MIME. Контейнер сообщения (*container message part*) может содержать закодированное изображение, аудиофайл или любые бинарные или текстовые данные, включая другие контейнеры. В приложениях для работы с электронной почтой они известны под названием "вложения" (*attachments*). На [Рисунке 10.1](#) показана структура сложного сообщения MIME. Прямоугольники со сплошными границами являются контейнерами, прямоугольники со штриховыми границами являются закодированными с помощью алгоритма Base64 бинарными данными, а прямоугольник с пунктирными границами - текстовым сообщением.

Контейнеры могут также быть произвольно вложенными; в этом случае они называются мультиконтейнерами (*multipart*) и фактически могут находиться на достаточно глубоком уровне вложенности. При этом любое сообщение электронной почты, независимо от его сложности, может быть представлено в виде дерева с единственным объектом сообщения на вершине. В рамках системы Mailman мы обычно рассматриваем сообщение как дерево объектов (*message object tree*) и передаем это дерево, ссылаясь на основной объект сообщения. На [Рисунке 10.2](#) показано дерево объектов сообщения с мультиконтейнерами, схематично изображенного на [Рисунке 10.1](#).

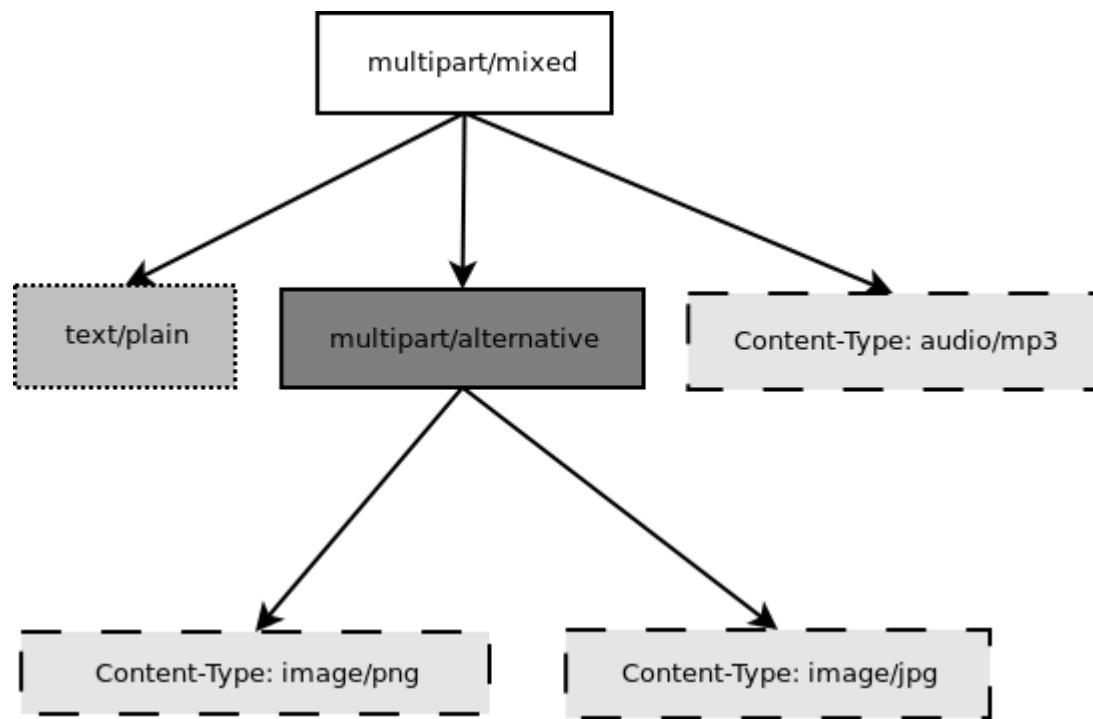


Рисунок 10.2: Дерево объектов для сложного MIME-типа сообщения электронной почты

Система Mailman практически всегда модифицирует оригинальное сообщение каким-либо образом. Иногда трансформации могут быть весьма незначительными, заключающимися в добавлении или удалении заголовков. Иногда мы полностью изменяем структуру дерева объектов сообщения, например, в том случае, когда фильтр содержимого удаляет определенные типы данных, такие, как текст в формате HTML, изображения или другие не текстовые данные. Система Mailman может даже удалить описание типа MIME "multipart/alternatives" в том случае, если сообщение содержит и текстовую часть и текстовые данные, использующие какой-либо тип разметки, или добавить дополнительные части сообщения, содержащие информацию о списке рассылки.

В общем случае система Mailman разбирает актуальное (*on the wire*) байтовое представление сообщения всего один раз в тот момент, когда сообщение передается системе. С этого момента система работает только с деревом объектов сообщения до тех пор, пока оно не готово для отправки на используемый почтовый сервер. В этот момент система Mailman преобразует дерево объектов обратно в байтовое представление. В ходе обработки сообщения Mailman сохраняет дерево объектов сообщения (с помощью модуля [pickle](#)) для хранения и последующего извлечения из файловой системы. Модуль pickle предоставляет функции, реализующие технологию языка Python для пре-

образования любого объекта Python, включая все его дочерние объекты, в последовательность байт (*сериализации*) и отлично подходит для оптимизации процесса обработки деревьев объектов сообщений электронной почты. Также доступна функция преобразования этой последовательности байт в активный объект (*десериализации*). Храня эти последовательности байт в файлах, программы на языке Python получают в свое распоряжение постоянное хранилище данных, требующее малых затрат ресурсов.

10.2. Список рассылки

Объект списка рассылки (*mailing list object*), очевидно, является еще одним основным объектом системы Mailman и большинство операций в рамках Mailman осуществляется именно со списком рассылки; эти операции:

- Участие в списке рассылки начинается после подписки на определенный список рассылки с использованием адреса пользователя.
- Списки рассылки имеют большое количество настроек конфигурации, хранящихся в базе данных и управляющих всеми аспектами функционирования приложения, начиная с привилегий размещения сообщений и заканчивая способом финальной модификации сообщений перед их доставкой.
- Списки рассылки имеют владельцев и модераторов, которые обладают большими возможностями по изменению аспектов функционирования списка, а также подтверждения или отклонения спорных сообщений.
- Каждый список рассылки имеет свой собственный архив.
- Пользователи отправляют новые сообщения в определенный список рассылки.

и так далее. Практически каждая операция в системе Mailman принимает ссылку на список рассылки в качестве аргумента - это фундаментальная концепция. Объекты списков рассылки были подвергнуты радикальным архитектурным изменениям в версии 3 системы Mailman для повышения их эффективности и гибкости.

Одним из ранних архитектурных решений, принятых John Viega, был способ представления объекта списка рассылки в рамках системы. Для представления этого центрального типа данных он выбрал класс Python с множеством базовых классов, каждый из которых реализовывал небольшую часть функций списка рассылки. Эти работающие совместно базовые классы, называемые смешанными классами (*mix-in classes*), были реализацией разумного способа организации кода, позволяющей просто добавлять необходимые совершенно новые функции. После подключения нового базового смешанного класса основной класс `MailList` может просто получить возможность выполнения каких-либо новых и замечательных действий.

Например, для добавления автоответчика в версию 2 системы Mailman был создан смешанный класс, который содержал специфические для данной функции данные. Данные должны автоматически инициализироваться при создании нового списка рассылки. Смешанный класс также предоставляет методы, необходимые для поддержки функции автоответчика. Эта структура стала более полезной при проектировании постоянного хранилища объекта списка рассылки `MailList`.

Другим ранним архитектурным решением, принятым John Viega, является использование модуля `pickle` языка Python для постоянного хранения данных состояния объекта `MailList`.

В версии 2 системы Mailman данные состояния объекта `MailList` сохраняются в файле с именем `config.pck`, который является простым представлением словаря объекта `MailList`, сформированным с использованием функций модуля `pickle`. Каждый объект языка Python имеет соответствующий атрибут в форме словаря с названием `__dict__`. Таким образом, сохранение объекта списка рассылки заключается в простом преобразовании его словаря `__dict__` с использованием функций модуля `pickle` в файл, а его загрузка - в чтении сохраненных данных из файла и реконструкции словаря `__dict__`.

Следовательно, когда смешанный класс добавляется для реализации каких-либо новых функций, все атрибуты смешанного класса автоматически преобразуются в файл и реконструируются соответствующим образом. Единственным дополнительным действием, которое нам приходится выполнять, является продержка версии схемы (*schema version number*) для возможности автоматического обновления устаревших объектов списков рассылок при добавлении новых атрибутов с помощью смешанных классов, так как сохраненное представление устаревших объектов MailList не будет содержать этих новых атрибутов.

Насколько бы не был удобен данный подход, архитектура смешанных объектов и постоянное хранилище данных с использованием функций модуля pickle не выдержали своего собственного веса. Администраторы сайтов часто просили предоставить способы доступа к конфигурационным переменным с использованием внешних систем, не основанных на технологиях языка Python. Но протокол представления данных модулем pickle является в полной мере специфичным для языка Python, поэтому объединение всех важных данных в файлах рассматриваемого формата в данном случае не является разумным. Также, ввиду того, что все данные состояния списка рассылки сохранялись в файле config.pck и система Mailman использовала множество процессов, которые должны были читать, модифицировать и записывать данные состояния списка рассылки, нам пришлось реализовать взаимные, работающие с NFS блокировки на основе файлов для того, чтобы быть уверенными в сохранности данных. В любой момент, когда какой-либо процесс системы Mailman хочет изменить данные состояния списка рассылки, он должен установить блокировку, записать измененные данные, после чего снять блокировку. Даже операции чтения могут потребовать перезагрузки соответствующего списку рассылки файла config.pck, так как некоторый другой процесс может изменить данные до выполнения операции чтения. Эта последовательность операций над данными списка рассылки была ужасно медленной и неэффективной.

По этим причинам система Mailman версии 3 хранит все данные в базе данных SQL. По умолчанию используется база данных SQLite 3, хотя эта настройка может быть просто изменена, так как в версии 3 системы Mailman применяется технология объектно-реляционного отображения (Object Relational Mapper) с названием Strom, поддерживающая большое количество баз данных. Поддержка PostgreSQL была добавлена всего лишь с помощью нескольких строк кода и администратор сайта может включить ее просто изменив одну из переменных конфигурации.

Другой, большей проблемой, присущей версии 2 системы Mailman, является разделенность списков рассылки. Обычно операции администрирования затрагивают множество списков рассылки, а иногда и все. Например, пользователь может захотеть временно заблокировать все свои подписки во время отпуска. Или администратор сайта может захотеть добавить какое-либо предупреждение в сообщение приветствия для всех списков рассылки своей системы. Даже простая операция по установлению того, на какой список подписан пользователь с заданным адресом, требует восстановления данных состояния каждого списка рассылки системы, так как информация о подписчиках также хранится в файле с именем config.pck.

Другая проблема заключалась в том, что каждый файл config.pck хорился в директории с именем, соответствующем названию списка рассылки, но приложение Mailman изначально проектировалось без учета возможности использования виртуальных доменов. Это обстоятельство привело к очень неприятной ситуации, в которой два списка рассылки не могли иметь одно и то же название в различных доменах. Например, если вы владеете и доменом example.com, и доменом example.org и при этом хотите, чтобы они были независимы и использовали отдельные списки рассылки с названиями support, вы не сможете реализовать эту идею в случае использования версии 2 системы Mailman без модификаций кода, использования не поддерживаемой точки вызова функций или определенных мер для обхода ограничения, которые позволяют принудительно устанавливать отличное название списка рассылки незаметно для пользователей и широко используются такими популярными сайтами, как SourceForge.

Эта проблема была решена в версии 3 системы Mailman путем изменения способа идентификации списков рассылки наряду с переносом всех данных в традиционную базу данных. Первичный ключ (*primary key*) таблицы списка рассылки является полностью определенным именем списка рассылки (*fully qualified list name*) или, как вы вероятнее воспримите, почтовым адресом. Следовательно, `support@example.com` и `support@example.org` представляются отдельными строками в таблице списков рассылки и могут просто существовать в рамках одной системы Mailman.

10.3. Обработчики

Сообщения перемещаются в рамках системы с помощью наборов независимых процессов, называемых обработчиками (*runners*). Изначально введенные в качестве инструмента для предсказуемой обработки всех файлов сообщений из очереди, найденных в определенной директории, на данный момент обработчики существуют в виде нескольких независимых постоянно функционирующих процессов, которые выполняют специфическую задачу и находятся под управлением ведущего процесса; более подробное описание будет приведено ниже. В том случае, если обработчик управляет файлами в директории, он называется обработчиком очереди (*queue runner*).

Система Mailman принципиально разрабатывается как однопоточное приложение даже при наличии возможностей параллельного выполнения процессов. Например, система Mailman может принимать сообщения от почтового сервера одновременно с отправкой сообщений принимающим сторонам, обработкой нагрузок или архивацией сообщения. Параллелизм в рамках системы Mailman достигается путем использования множества процессов в виде этих самых обработчиков. Например, существует обработчик очереди входящих сообщений (*incoming queue runner*), задачей которого является прием (или отклонение) сообщений от используемого сервера электронной почты. Также существует обработчик очереди исходящих сообщений (*outgoing queue runner*), задачей которого является взаимодействие с используемым SMTP-сервером для отправки сообщений ко-нечным адресатам. Наряду с описанными существуют обработчики очереди архивирования (*archiver queue runner*), очереди уведомлений (*bounce processing queue runner*), обработчик очереди пересылки сообщений серверу NNTP (*queue runner for forwarding messages*), обработчик создания каталога (*runner for composing digests*) и некоторые другие обработчики. Обработчики, не управляющие очередями включают обработчик с реализацией локального протокола пересылки почты ([Local Mail Transfer Protocol](#)) и обработчик с реализацией административного HTTP-сервера.

Каждый обработчик очереди ответственен за отдельную директорию, т.е., за свою очередь. Хотя обычная система Mailman может превосходно работать, выделяя по одному процессу на очередь, мы можем применить сложный алгоритм для параллельного исполнения задач в рамках отдельной директории очереди, не используя при этом каких-либо типов взаимодействий и блокировок. Секретом успеха является способ именования файлов в директории очереди.

Как упоминалось ранее, каждое сообщение, которое перемещается в системе также сопровождается словарем для метаданных, который накапливает данные состояния и позволяет независимым компонентам системы Mailman взаимодействовать друг с другом. Библиотека `pickle` языка Python позволяет производить прямое и обратное преобразование множества объектов в один файл, поэтому мы можем преобразовать в один и тот же файл и дерево объектов сообщения и словарь метаданных.

В рамках системы Mailman существует основной класс с именем `Switchboard`, который предоставляет интерфейс для выполнения операций помещения в очередь (т.е. записи) и извлечения из очереди (т.е. чтения) дерева объектов сообщения и словаря метаданных в отношении файлов из директории определенной очереди. Каждая директория очереди имеет как минимум один соответствующий ей экземпляр класса `Switchboard` и каждый экземпляр обработчика очереди имеет по одному экземпляру класса `Switchboard`.

Все созданные с использованием библиотеки pickle файлы имеют расширение .pck, хотя вы также можете обнаружить в очереди файлы с расширениями .bak, .tmp, и .psv. Они используются для реализации двух священных принципов функционирования системы Mailman: ни один из файлов не должен быть потерян и ни одно из сообщений не должно быть доставлено более чем один раз. Но обычно система функционирует в нормальном режиме, поэтому эти файлы могут быть обнаружены очень редко.

Как было показано, для особо загруженных сайтов система Mailman предоставляет возможность запуска более чем одного процесса обработчика для каждой из директорий очередей в полностью параллельном режиме без взаимодействия между ними или необходимости блокировок для обработки файлов. Этого удается добиться путем использования хэшей SHA1 для именования файлов, после чего разрешения отдельному обработчику очереди обрабатывать только определенный участок диапазона хэш-значений. Таким образом, если сайту необходимо использовать два обработчика для очереди уведомлений (*bounces queue*), один обработчик будет обрабатывать файлы из верхней половины диапазона хэш-значений, а другой будет обрабатывать файлы из нижней половины диапазона хэш-значений. Хэши рассчитываются с использованием данных дерева объектов сообщения из файла, имени списка рассылки, для которого предназначено сообщение и метки времени. Хэши SHA1 эффективно распределены и, следовательно, в среднестатистической директории с двумя обработчиками очередей у каждого процесса будет примерно одинаковый объем работы. А так как диапазон хэш-значений может быть статически разделен, эти процессы могут работать в одной директории очереди без необходимости вмешательства в работу друг друга и взаимодействия.

Существует интересное ограничение данного алгоритма. Так как алгоритм разделения ставит в соответствие каждому обработчику одни или несколько битов хэша, количество обработчиков для каждой директории очереди должно быть кратным 2. Это значит, что могут использоваться 1, 2, 4 или 8 процессов обработчиков для каждой очереди, но, например, не 5. На практике это ограничение никогда не приводило к проблемам, так как только нескольким сайтам может потребоваться более 4 обработчиков для работы в условиях их нагрузки.

Существует другой побочный эффект использования этого алгоритма, который приводил к проблемам в период раннего проектирования этой системы. Несмотря на непредсказуемость процесса доставки сообщений электронной почты в общем случае, для пользователя является наиболее удобной обработка файлов очереди в последовательности FIFO, таким образом, чтобы направляемые в список рассылки ответы отсылались в относительно хронологическом порядке. Игнорирование этого правила может привести в замешательство участников списка рассылки. Но использование хэшей SHA1 в качестве имен файлов препятствует использованию меток времени, при этом следует избегать вызова функции `stat()` в отношении файлов очереди по причинам, связанным с производительностью, а также распаковки содержимого сообщения (т.е. чтения метки времени из метаданных).

Решение в рамках системы Mailman заключалось в расширении алгоритма именования файлов для включения в состав имени префикса с меткой времени в форме числа, отражающего количество секунд, прошедших с начала эпохи (т.е. <метка времени>+<хэш sha1>.pck). Каждый обход очереди обработчик начинает с вызова метода `os.listdir()`, который возвращает список всех файлов в директории очереди. После этого для каждого файла производится разбор имени файла и игнорируются все имена файлов, хэши SHA1 которых не соответствуют диапазону хэшей обработчика. После этого обработчик сортирует оставшиеся файлы на основе данных из части имен, содержащей метку времени. Утверждение о том, что при использовании множества обработчиков очередей, каждый из которых обрабатывает определенный диапазон хэш-значений, могут возникнуть проблемы с распределением файлов между параллельно функционирующими обработчиками, является истинным, но на практике распределение на основе меток времени достаточно для удовлетворения ожиданий конечных пользователей в плане последовательной доставки сообщений.

На практике этот метод работал очень хорошо как минимум в течение 10 лет с проводимыми время от времени исправлениями незначительных ошибок или доработками для использования в частных случаях и в условиях возникновения ошибок. Это одна из наиболее стабильных частей системы Mailman, которая была портирована практически без изменений из Mailman 2 в Mailman 3.

10.4 Ведущий обработчик

При использовании всех описанных процессов обработчиков системе Mailman требуется простой способ их запуска и остановки; именно с этой целью был создан процесс ведущего обработчика, следящего за состоянием других обработчиков. У него должна быть возможность управления и обработчиками очередей, и обработчиками, которые не управляют очередями. Например, в версии 3 системы мы принимаем сообщения от используемого почтового сервера посредством протокола LMTP, который аналогичен протоколу SMTP, но позволяет выполнять только локальную доставку почтовых сообщений и, следовательно, может быть значительно упрощен, так как при его использовании не требуется обрабатывать нестандартные ситуации, возникающие в ходе доставки почты с использованием ненадежного соединения с Интернет. Обработчик протокола LMTP просто прослушивает порт, ожидая от используемого почтового сервера соединения и отправки байтового потока. После этого он преобразует этот байтовый поток в дерево объектов сообщения, создает начальный словарь метаданных и добавляет сообщение в обрабатываемую директорию очереди.

Система Mailman также использует обработчик, который прослушивает другой порт и обрабатывает REST-запросы, переданные по протоколу HTTP. Этот процесс вообще не затрагивает файлы очереди.

Стандартная работающая система Mailman может использовать восемь или десять процессов, причем все они должны быть остановлены и запущены корректно и согласованно. Они также могут внезапно аварийно завершаться; например, в случае, когда ошибка в Mailman приводит к неожиданному исключению. Когда это происходит доставляемое сообщение блокируется (*shunted*) и помещается в хранилище вместе с данными состояния системы в момент возникновения исключения, сохраненными в метаданных сообщения. Эта операция позволяет быть уверенным в том, что необработанное исключение не приведет к множеству повторных отправок сообщения. Теоретически администратор сайта, работающего под управлением системы Mailman, может устраниТЬ проблему, после чего разблокировать (*unshunt*) вызвавшие проблемы сообщения для повторной доставки с момента остановки процесса. После блокировки проблемного сообщения ведущий обработчик перезапускает аварийно завершивший работу обработчик очереди, который начинает обработку оставшихся сообщений из своей очереди.

При запуске ведущий обработчик исследует файл конфигурации для установления того, какое количество и какие типы дочерних обработчиков должны быть запущены. В случае обработчиков для поддержки протоколов LMTP и REST обычно используется по одному процессу. В случае обработчиков очередей, как было сказано выше, может быть запущено кратное двум количество параллельно выполняющихся процессов. Ведущий обработчик использует вызовы `fork()` и `exec()` для запуска всех обработчиков, необходимость которых заявлена в файле конфигурации, передавая соответствующие параметры командной строки каждому из них (т.е., указывая подпроцессу на то, с каким диапазоном хэш-значений ему следует работать). После этого ведущий обработчик блокируется в бесконечном цикле, ожидая завершения работы одного из своих дочерних процессов. Он отслеживает идентификаторы каждого из дочерних процессов вместе с количеством перезапусков каждого дочернего процесса. Этот подсчет позволяет избежать катастрофической ошибки, приводящей к непрерывному каскаду перезапусков. Существует переменная конфигурации, которая устанавливает разрешенное количество перезапусков, по истечении которого сообщение об ошибке заносится в системный журнал и обработчик больше не перезапускается.

При завершении работы дочернего процесса ведущий обработчик рассматривает код завершения процесса и сигнал, с помощью которого был завершен подпроцесс. Каждый процесс обработчика устанавливает функции обработки сигналов со следующими семантиками:

- SIGTERM: умышленное завершение подпроцесса. Перезапуск не должен производиться. Сигнал SIGTERM является тем сигналом, который отправляется процессом `init` для завершения работы процессов при изменении уровней выполнения, а также является тем сигналом, который отправляет система Mailman для завершения работы подпроцесса.
- SIGINT: также используется для умышленного завершения подпроцесса и является тем сигналом, который используется при использовании сочетания клавиш control-C в командной оболочке. Обработчик не перезапускается.
- SIGHUP: сообщает процессу о необходимости закрытия и повторного открытия файлов журнала без прерывания работы. Сигнал используется для ротации файлов журнала.
- SIGUSR1: изначальная остановка подпроцесса с разрешением его перезапуска ведущим обработчиком. Используется для реализации команды `restart` сценариев инициализации.

Ведущий обработчик также принимает четыре описанных выше сигнала, но не выполняет никаких действий, кроме рассылки их всем подпроцессам. Следовательно, если вы отправили сигнал SIGTERM ведущему обработчику, все подпроцессы получат сигнал SIGTERM и завершат работу. Ведущий обработчик получит информацию о том, что обработчик завершил работу после приема сигнала SIGTERM, а также о том, что это завершение работы было умышленным, поэтому он не перезапустит обработчик.

Для того, чтобы быть уверенным в том, что только один процесс ведущего обработчика выполняется в каждый момент времени, этот процесс устанавливает блокировку с временем существования около дня или половины дня. Ведущий обработчик устанавливает функцию обработки сигнала SIGALRM, которая позволяет раз в день обновлять блокировку. Так как время жизни блокировки превышает интервал ее обновления, время действия блокировки никогда не истечет и блокировка не станет неработоспособной во время функционирования системы Mailman, конечно же, за исключением случаев, когда система аварийно завершит работу или выполнение процесса ведущего обработчика будет завершено с использованием не обрабатываемого сигнала. Для этих случаев интерфейс командной строки процесса ведущего обработчика предусматривает параметр, позволяющий игнорировать блокировку с истекшим сроком действия.

Таким образом, мы переходим к заключительной части описания ведущего обработчика, а именно, описанию его интерфейса командной строки. Сам сценарий ведущего обработчика может принимать очень малое количество параметров командной строки. Этот сценарий, как и сценарии обработчиков очередей, умышленно упрощены. Это утверждение не было справедливым при рассмотрении версии 2 системы Mailman, в которой сценарий ведущего обработчика был достаточно сложен и пытался выполнять большое количество действий, что в большей степени затрудняло его понимание и отладку. В версии 3 системы Mailman реальный интерфейс командной строки для ведущего обработчика находится в сценарии `bin/mailman`, являющимся типом мета-сценария и содержащим множество подкоманд, оформленных в стиле, набравшем популярность вместе с такими программами, как Subversion. Это решение позволяет сократить количество программ, которые должны быть установлены в директорию, заданную переменной PATH вашей оболочки. В сценарии `bin/mailman` имеются подкоманды для запуска, остановки и перезапуска ведущего обработчика, также, как и всех его подпроцессов, а также команды для принудительного повторного открытия всех файлов журналов. Подкоманда `start` использует вызовы функций `fork()` и `exec()` процессом ведущего обработчика, в то время, как все остальные просто отправляют соответствующий сигнал ведущему серверу, который затем распространяет этот сигнал среди своих подпроцессов, как описано выше. Это улучшенное разделение ответственности упрощает понимание роли каждого отдельного программного компонента.

10.5. Правила, звенья и цепочки

Размещение сообщений в списке рассылки состоит из нескольких фаз, от момента первого приема сообщения до момента отправки сообщения участникам списка рассылки. В версии 2 системы

Mailman каждый этап обработки сообщения был представлен обработчиком сообщения (*handler*), а наборы обработчиков объединялись в канал (*pipeline*). Следовательно, в тот момент, когда сообщение поступает в систему, Mailman в первую очередь установит, какой канал будет использован для его обработки, после чего каждый обработчик сообщения канала будет вызван по очереди. Некоторые обработчики сообщений выполняют функции модерации (т.е., функции, заключающиеся в ответе на вопрос: "Разрешено ли данному человеку размещать сообщения в списке рассылки?"), другие выполняют функции модификации сообщений (в данном случае вопрос звучит так: "Какие заголовки следует убрать или добавить?"), а другие - копируют сообщение в другие очереди. Несколько примеров использования обработчиков сообщений последнего типа:

- Сообщение, принятое для размещения, должно быть скопировано в очередь `archiver` на каком-либо этапе, таким образом обработчик очереди сможет добавить сообщение в архив.
- Копия сообщения в конечном счете должна быть помещена в очередь `outgoing` для того, чтобы была возможность передать ее используемому почтовому серверу, который несет персональную ответственность за доставку сообщения участнику списка рассылки.
- Копия сообщения должна быть помещена в каталог для использования людьми, которые хотят время от времени получать сообщения от списка рассылки вместо получения каждого сообщения сразу же после его отправки в список рассылки.

Архитектура, использующая канал из обработчиков сообщений, продемонстрировала достаточную эффективность. Она предоставляет простой способ, который люди могут использовать для расширения и изменения возможностей системы Mailman с целью выполнения специфических операций. Интерфейс обработчика сообщений был достаточно простым, что было причиной для реализации новых обработчиков сообщений, при этом следовало просто убедиться в том, что он добавлен в необходимый канал в нужном месте для выполнения специфической операции.

Единственной проблемой данного подхода является тот факт, что смешивание функций модерации и модификации сообщений в рамках одного канала может привести к проблемам. Обработчики должны быть установлены в канале в определенной последовательности, иначе результаты обработки сообщений могут быть непредсказуемыми и нежелательными. Например, если обработчик сообщения добавит заголовки `List-*`, описанные стандартом [RFC 2369](#), после того, как другой обработчик сообщения скопирует его в хранилище каталога, подписчикам, получающим каталоги, будут отправлены некорректные копии сообщений из списка рассылки. В различных случаях может оказаться выгодной модерация сообщения до или после его модификации. В версии 3 системы Mailman операции модерации и модификации были разделены и реализованы в рамках отдельных подсистем для лучшего контроля над последовательностью их выполнения.

Как было описано ранее, обработчик для поддержки протокола LMTP производит разбор входящего байтового потока, преобразует его в дерево объектов сообщения и создает начальный словарь метаданных сообщения. После этого он перемещает сообщения в одну из других директорий очередей. Некоторые сообщения могут быть *email-командами* (*email commands*) (т.е., командами для создания или удаления подписки на список рассылки, для получения автоматизированной справки, и.т.д.), обрабатываемыми в рамках отдельной очереди. Большинство сообщений являются сообщениями для размещения в списке рассылки, которые помещаются в очередь входящих сообщений (*incoming queue*). Обработчик очереди входящих сообщений обрабатывает каждое сообщение последовательно в цепочке (*chain*), состоящей из любого количества звеньев (*links*). Существует встроенная цепочка, которую использует большинство списков рассылки, но даже это возможно изменить путем изменения параметра конфигурации.

[Рисунок 10.3](#) иллюстрирует стандартный набор цепочек версии 3 системы Mailman. Каждое звено в цепочке показано с помощью прямоугольника со скругленными краями. Встроенная цепочка отличается тем, что в ней в отношении входящих сообщений применяются начальные правила модерации, а также в этой цепочке каждое звено ассоциировано с правилом (*rule*). Правила являются простыми фрагментами кода, которым передается три стандартных параметра: объект списка рассылки, дерево объектов сообщения и словарь метаданных сообщения. Правила не предназначены для модификации сообщения; они просто принимают решение и возвращают логическое значение,

отвечая на вопрос: "Выполняется ли правило или нет?". Правила также могут записывать информацию в словарь метаданных сообщения.

На рисунке сплошные линии со стрелками отображают направления потока сообщений в случае выполнения правила, а пунктирные линии со стрелками отображают направления потока сообщений в случае невыполнения правила. Результат проверки каждого правила записывается в словарь метаданных сообщения, поэтому впоследствии система Mailman будет располагать информацией (и сможет сформировать отчет) о том, какие конкретно правила были выполнены, а какие - нет. Штриховые линии со стрелками указывают на безусловные перемещения сообщений, не связанные с тем, выполнилось ли правило или нет.

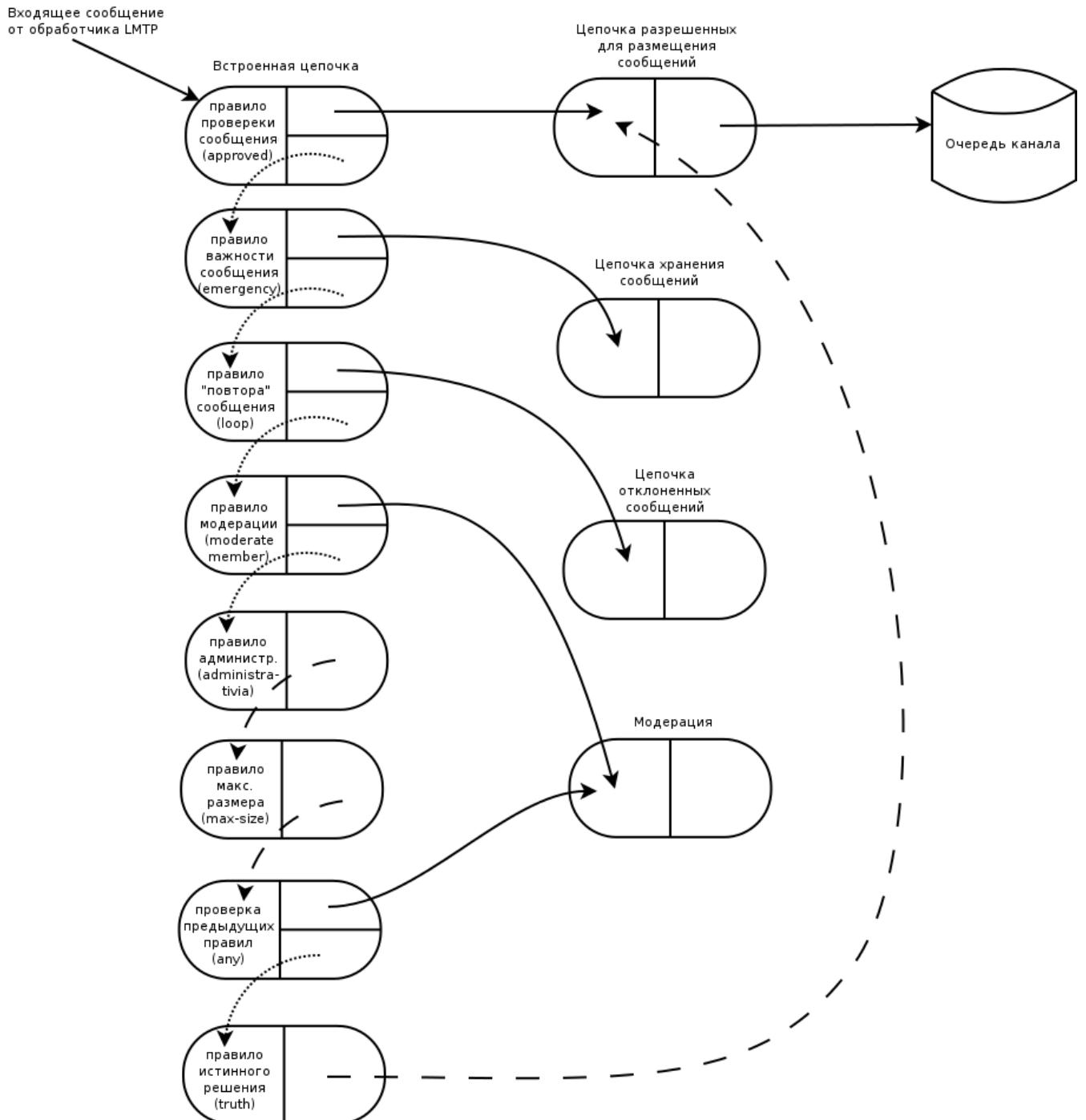


Рисунок 10.3: Упрощенный вид стандартных цепочек с их звеньями

Важно отметить, что сами правила не выполняют действий на основании результата. Во встроенной цепочке каждое звено ассоциировано с действием (action), которое выполняется в случае выполнения правила. Например, когда выполняется правило "повтора" (определенное, что это со-

общение уже встречалось ранее в списке рассылки), сообщение немедленно перемещается в цепочку "отклоненных сообщений", из которой оно будет удалено по истечении некоторого периода хранения. Если правило "повтора" не выполняется, то следующее звено в цепочке будет обрабатывать это же сообщение.

На [Рисунке 10.3](#) звенья, ассоциированные с правилами административной проверки ("administrivia"), проверки максимального размера ("max-size") и истинного решения ("truth") не возвращают логических результатов. В случае первых двух правил такая реализация объясняется тем, что их действие *отсрочено*, поэтому они просто записывают результат сравнения и обработка сообщения продолжается следующим звеном. После этого правило проверки любого значения ("any") проверяет, выполнено ли любое из предыдущих правил. Таким образом система Mailman может сообщить о всех причинах запрета на размещение сообщения, вместо единственной первой причины. Существует еще несколько таких правил, которые не были отображены на рисунке с целью упрощения представления.

Правило истинного решения ("truth") немного отличается. Оно обычно ассоциируется с последним звеном в цепочке и всегда выполняется. В комбинации с предпоследним в цепочке правилом проверки любого значения ("any"), затрагивающим результаты выполнения правил в отношении всех ранее обработанных сообщений, последнее звено может быть уверенно в том, что любое достигшее его сообщение должно быть размещено в списке рассылки, поэтому оно безусловно перемещается в цепочку разрешенных для размещения сообщений ("accepted" chain).

Существует несколько других особенностей процесса обработки цепочек, не описанных в данной главе, но архитектура этой части системы является очень гибкой и расширяемой, поэтому в ее рамках может быть реализован практически любой метод обработки сообщений и администраторы сайтов могут изменять и расширять набор функций правил, звеньев и цепочек.

Что случится с сообщением после попадания в цепочку разрешенных для размещения сообщений ("accept" chain)? Сообщение, которое теперь считается подходящим для списка рассылки направляется в очередь канала (*pipeline queue*) для выполнения некоторых модификаций перед доставкой принимающим сообщения подписчикам списка рассылки. Этот процесс более подробно описан в следующем разделе.

Цепочка хранения сообщений ("hold" chain) помещает сообщение в специальное хранилище для ознакомления с ним человека, выполняющего функции модератора. Цепочка модерации ("moderation" chain) выполняет небольшое количество дополнительных операций для формирования решения по поводу того, должно ли сообщение быть принято, заблокировано для утверждения модератором, не принято или отклонено. Для того, чтобы избежать чрезмерного усложнения диаграммы, очередь отклоненных сообщений, которая используется для возвращения сообщений отправителям, не отображена.

10.6. Обработчики сообщений и каналы

Как только сообщение проходит через все звенья и преодолевает все проверки выполнения правил, получая разрешение на размещение в списке рассылки, оно должно быть дополнительно обработано перед тем, как будет доставлено конечным адресатам. Например, некоторые заголовки могут быть добавлены или удалены, а также некоторые сообщения могут подвергнуться дополнительным модификациям для предоставления важных предупреждений или информации, такой, как информация том, как покинуть список рассылки. Эти модификации выполняются в рамках канала, содержащего последовательность обработчиков сообщений. Аналогично обработчикам сообщений в звеньях при использовании правил, функции каналов и обработчиков сообщений также могут быть расширены, но существует набор встроенных каналов, предназначенных для стандартного использования. Обработчики сообщений имеют такой же интерфейс, как и правила, принимающий объект списка рассылки, объект сообщения и словарь метаданных. Однако, в отличие от

правил, обработчики сообщений могут дополнять и модифицировать сообщение. [Рисунок 10.4](#) иллюстрирует стандартный канал и набор обработчиков сообщений (некоторые обработчики сообщений исключены для упрощения).

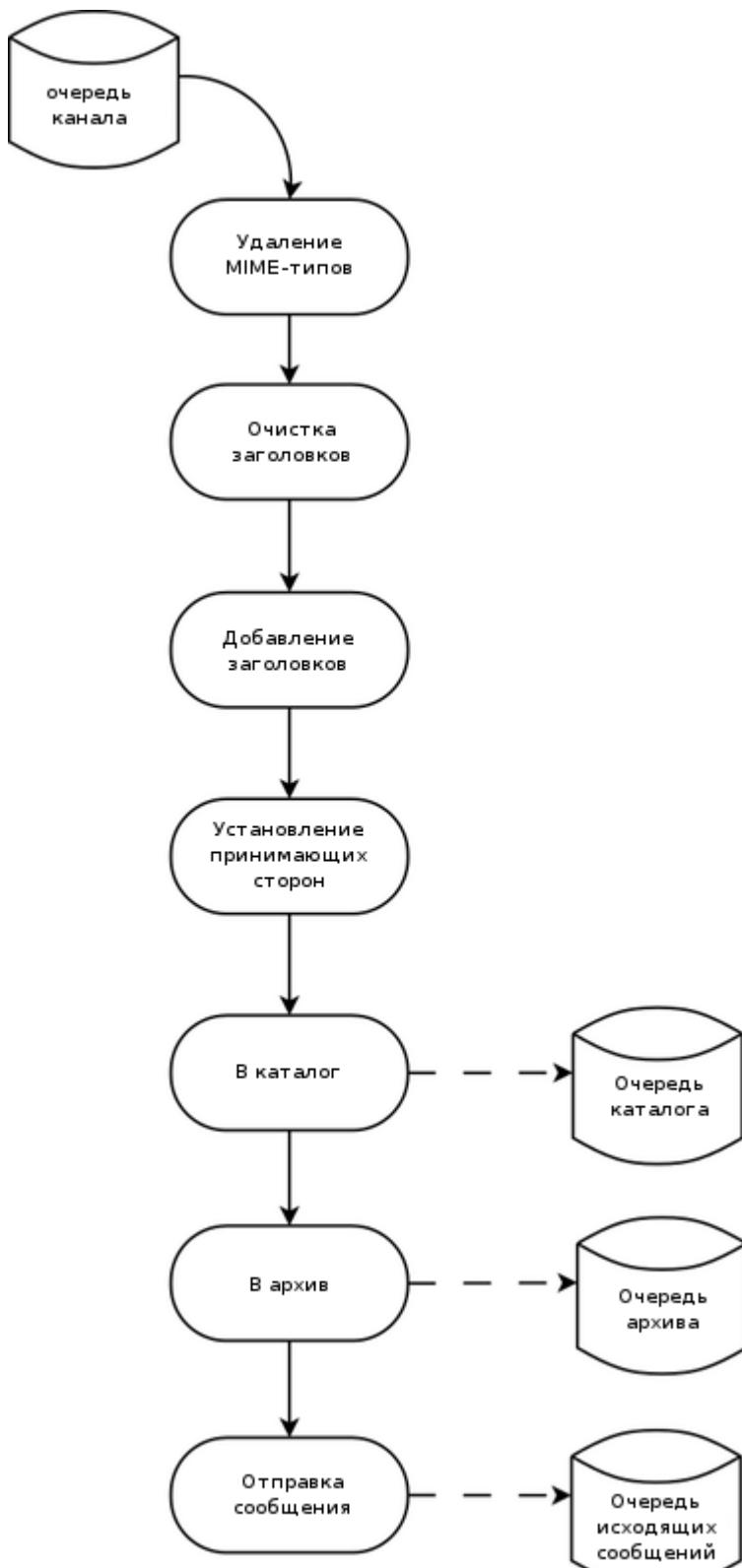


Рисунок 10.4: Очередь обработчиков сообщения в рамках канала

Например, отправленное сообщение должно иметь добавленный заголовок `Precedence:`, который сообщает другим автоматически функционирующими программным компонентам о том, что сообщение пришло из списка рассылки. Этот заголовок является стандартом де-факто, позволяющим предотвратить отправку ответов на сообщения из списка рассылки при использовании автоматизированных программ. Добавление этого заголовка (наряду с другими модификациями заголов-

ков) осуществляется с помощью обработчика сообщений для добавления заголовков ("add header" handler). В отличие от правил, порядок использования обработчиков сообщений в общем случае не существен и сообщения всегда проходят через все обработчики сообщений канала.

Некоторые обработчики сообщений отправляют копии сообщения в другие очереди. Как показано на [Рисунке 10.4](#), существует обработчик сообщения, создающий копию сообщения для подписчиков, которые хотят принимать каталоги сообщений. Копии также отправляются в очередь архива для последующего помещения в архив списков рассылки. Наконец, сообщение копируется в очередь исходящих сообщений для окончательной доставки подписчикам списка рассылки.

10.7. VERP

Аббревиатура VERP расшифровывается как [Variable Envelope Return Path](#) и используется для обозначения широко известной техники, используемой списками рассылок для однозначного определения реального адреса принимающей стороны. Когда адрес в списке рассылки перестает быть активным, почтовый сервер принимающей стороны отправит уведомление об этом отправителю сообщения. В случае списки рассылки вам потребуется сделать так, чтобы это уведомление было отправлено в список рассылки, а не автору оригинального сообщения; автор не сможет сделать ничего с уведомлением и хуже того, отправка сообщения назад автору может раскрыть информацию о том, кто подписан на список рассылки. Когда список рассылки получает уведомление, однако, он может выполнить какое-либо полезное действие, такое, как отключение адреса из уведомления или удаление подписки на список рассылки, использующей этот адрес.

При этом существуют две основные проблемы. Во-первых, даже с учетом того, что существует стандартный формат описанных выше уведомлений (называемый [delivery status notifications](#)), многие используемые серверы электронной почты не соблюдают его. Вместо этого основная часть их уведомлений может содержать любой объем сложно разбираемого программно текста, что затрудняет их автоматическую обработку. Фактически система Mailman использует библиотеку, содержащую множество данных для эвристического анализа уведомлений, сформированную из данных реально принятых уведомлений за период 15 лет существования Mailman.

Во-вторых, представим ситуацию, при которой участник списка рассылки использует несколько перенаправлений сообщений. Участница может быть подписана на список рассылки с использованием ее адреса `anne@example.com`, но сообщения с него могут перенаправляться на адрес `person@example.org`, с которого сообщения также могут перенаправляться на адрес `me@example.net`. Когда последний сервер `example.net` получает сообщение, он обычно просто отправляет сообщение, указывающее на то, что адрес `me@example.net` более не действителен. Но сервер Mailman, отправивший сообщение, знает только об адресе участника `anne@example.com`, поэтому уведомление о недействительном адресе `me@example.net` не будет содержать адреса подписчика и Mailman проигнорирует его.

В данном случае используется техника VERP, которая эксплуатирует фундаментальное [требование к протоколу SMTP](#) о предоставлении возможности однозначного определения адреса назначения путем возвращения таких уведомлений отправителю сообщения (*envelope sender*). Эта операция осуществляется не с использованием поля `From:` сообщения, а фактически с использованием значения `MAIL FROM`, устанавливаемого во время диалога SMTP. Эта информация сохраняется в ходе доставки сообщения и конечный почтовый сервер должен, в соответствии со стандартами, отправлять уведомления на полученный адрес. Система Mailman использует этот факт для кодирования оригинального адреса назначения в качестве значения `MAIL FROM`.

Если сервер Mailman использует адрес `mylist@example.org`, то закодированный адрес отправителя сообщения для размещения в списке рассылки с помощью технологии VERP, отправляемый на адрес `anne@example.com`, будет следующим:

`mylist-bounce+anne@example.com@example.org`

В данном случае символ + используется для отделения локального адреса, причем это форматирование поддерживается большинством современных почтовых серверов. Таким образом, когда сообщение возвратится, оно на самом деле будет доставлено на адрес `mylist-bounce@example.com`, но в заголовке `To:` будет находиться закодированный с использованием техники VERP адрес получателя. После этого система Mailman может произвести разбор этого заголовка `To:` для декодирования оригинального адреса назначения в виде `anne@example.com`.

Хотя техника VERP и является очень мощным инструментом для фильтрации некорректных адресов и недопущения их попадания в список рассылки, она имеет один потенциально важный недостаток. Использование техники VERP требует от системы Mailman отправки только одной копии сообщения для каждой принимающей стороны. Без техники VERP система Mailman могла отсылать наборы идентичных копий исходящих сообщений множеству принимающих сторон, экономя тем самым общую пропускную способность канала и время обработки сообщений. Но техника VERP требует использования уникального значения `MAIL FROM` для каждой принимающей стороны, а единственным способом удовлетворения этого требования является отправка уникальных копий сообщения. В общем случае это приемлемый компромисс и, фактически, вместе с отправкой этих индивидуальных сообщений для задействования техники VERP, система Mailman также сможет сделать множество полезных вещей в любом случае. Например, она может встраивать строку URL в заключительную часть сообщения, сформированную для каждого из подписчиков и позволяющую использовать прямую ссылку для закрытия подписки. Вы можете представить множество различных типов операций обработки сообщений с целью модификации текста сообщения для каждого индивидуального подписчика.

10.8. REST

Одно из ключевых архитектурных изменений в версии 3 системы Mailman заключается в удовлетворении запроса, предъявляемого к ней в течение многих лет: предоставления упрощенного способа интеграции Mailman со сторонними системами. Когда я был нанят компанией Canonical, являющейся корпоративным спонсором проекта Ubuntu в 2007 году, моя работа изначально заключалась в добавлении возможности использования списков рассылки в Launchpad, хостинг-платформу для совместной работы над программными проектами. Я знал, что версия 2 системы Mailman подходила для решения задачи, но при этом требовалось задействовать пользовательский веб-интерфейс системы Launchpad вместо стандартного пользовательского веб-интерфейса системы Mailman. Так как списки рассылки системы Launchpad практически всегда использовались в качестве дискуссионных списков рассылки, нам хотелось добиться сокращения различий в методах управления ими. Администраторы списков рассылки не должны иметь в распоряжении избыточное количество параметров конфигурации, доступных на стандартном сайте под управлением системы Mailman, поэтому оставалось дать ответ на вопрос о том, какие параметры могут понадобиться и будут представлены в пользовательском интерфейсе системы Launchpad.

В тот момент система Launchpad не являлась свободным программным обеспечением (это изменилось в 2009 году), поэтому нам пришлось проектировать уровень интеграции таким образом, чтобы код системы Mailman версии 2, распространявшийся в соответствием с условиями лицензии GPLv2, не затрагивал код Launchpad. Это привело к выработке ряда архитектурных решений в ходе проектирования архитектуры уровня интеграции, которые были достаточно неочевидными и отчасти неэффективными. Так как на данный момент система Launchpad является свободным программным обеспечением, распространяемым в соответствии с условиями лицензии AGPLv3, эти решения на сегодняшний день не являются необходимыми, но работа над ними позволила получить полезные знания о том, как система Mailman без пользовательского интерфейса может быть интегрирована в состав сторонних систем. Стал очевидным тот факт, что основная часть системы, эффективно и надежно реализующая операции для работы со списком рассылки, может работать

под управлением любого типа веб-интерфейса, включая интерфейсы на основе Zope, Django или PHP, а также вообще без пользовательского веб-интерфейса.

В тот момент для реализации этой идеи был доступен ряд технологий, при этом фактически интеграция систем Mailman и Launchpad основана на использовании протокола XMLRPC. Но протокол XMLRPC имеет ряд проблем, что делает его не идеальным протоколом.

В версии 3 системы Mailman была применена модель передачи состояния представления (Representational State Transfer - REST) для внешнего административного управления. Модель REST основана на протоколе HTTP и стандартном формате представления объектов системы Mailman, JSON. Эти протоколы используются повсеместно и отлично поддерживаются большинством разнообразных языков программирования и окружений, делая интеграцию Mailman со сторонними системами достаточно простой. Модель REST отлично подходила для использования совместно с версией 3 системы Mailman, а сейчас большинство функций системы доступно при использовании REST API.

Это мощная парадигма, которую должно использовать большее количество приложений: предоставить основную часть, которая качественно реализует базовые функции, предоставляя REST API для управления ею. REST API предоставляет дополнительный способ интеграции с системой Mailman, в дополнение к использованию интерфейса командной строки или разработки кода на языке Python для доступа к внутреннему API. Эта архитектура является очень гибкой и может быть использована и интегрирована такими способами, которые находятся за гранью стандартного видения системных архитекторов.

Такая архитектура позволяет не только вести разработку большим количеством способов, но даже проектировать и реализовывать официальные компоненты системы. Например, новый официальный пользовательский веб-интерфейс в версии 3 системы Mailman технически является отдельным проектом со своей кодовой базой, развивающимся в основном опытными веб-дизайнерами. Эти выдающиеся разработчики имеют возможность принимать решения, изменять дизайн и запускать реализации веб-интерфейса, не испытывая затруднений из-за хода разработки основного кода системы. В ходе разработки пользовательского веб-интерфейса отправляются запросы добавления необходимых функций в основную часть кода и разрешения доступа к ним с использованием REST API, но разработчикам веб-интерфейса не нужно ждать реализации необходимых функций, так как они могут использовать прототип сервера и продолжать экспериментировать и разрабатывать пользовательский веб-интерфейс в то время, как будет дорабатываться основная часть кода.

Мы планируем использовать REST API для множества других вещей, включая возможность добавления стандартных операций в сценарии и интеграцию с серверами IMAP или NNTP для предоставления альтернативной возможности доступа к архивам.

10.9. Интернационализация

Приложение GNU Mailman было одним из первых приложений на языке Python, поддерживающим функции интернационализации. Конечно же, так как система Mailman не всегда модифицирует содержимое проходящих через нее сообщений электронной почты, эти сообщения могут использовать любой язык в соответствии с выбором автора. Однако, при прямом взаимодействии с Mailman либо с использованием веб-интерфейса, либо с использованием команд из сообщений электронной почты, пользователи предпочтут использовать их родной язык.

Система Mailman впервые использовала множество технологий интернационализации, предлагаемых языком Python, но более запутанным способом, чем большинство приложений. В стандартном окружении рабочего стола язык выбирается при входе пользователя в систему и остается неизменным в течение сессии. Однако, Mailman является серверным приложением, поэтому оно должно иметь возможность работать с множеством языков вне зависимости от языка, с кото-

рым работает система. Фактически система Mailman должна определять контекст языка (*language context*), с учетом которого должен отправляться ответ, и переводить свой текст на этот язык. Иногда для формирования ответа может потребоваться использовать множество языков; например, если уведомление от пользователя из Японии должно быть перенаправлено в список рассылки администраторов, которые разговаривают по-немецки, по-итальянски и по-кATALОНСКИ.

Повторю, что система Mailman впервые использовала ключевые технологии языка Python, предназначенные для обработки таких сложных контекстов языка, как эти. Она использует библиотеку, управляющую стеком языков, которые могут быть добавлены или извлечены в форме изменений контекста, даже в случае обработки единственного сообщения. Также реализуется продуманная схема изменения шаблонов ответов на основе настроек сайта, настроек владельца списка рассылки и выбранного языка. Например, если владелец списка рассылки хочет использовать заданный шаблон ответа для одной из рассылок, но только для пользователей из Японии, он может разместить определенный шаблон в определенном месте файловой системы, после чего более общие настройки системы не будут учитываться.

10.10. Выученные уроки

Хотя в данной главе и представлен обзор архитектуры версии 3 системы Mailman, а также подробный обзор процесса эволюции этой архитектуры в течение 15 лет существования программного продукта (после трех значительных переработок кода), существует много интересных архитектурных решений, примененных в рамках Mailman, которые мне не удалось затронуть. Они включают в себя подсистему конфигурации, инфраструктуру тестирования, уровень для работы с базой данных, программное использование формальных интерфейсов, архивирование, стили списков рассылки, команды из сообщений электронной почты и интерфейс командной строки, а также вопросы интеграции с используемым сервером электронной почты. Свяжитесь с нами с помощью [списка рассылки разработчиков](#) системы Mailman в том случае, если вам интересны эти вопросы и вы хотите получить более подробное описание.

Ниже приведен список уроков, усвоенных нами в ходе переработки кода популярной, признанной и стабильной части экосистемы приложений с открытым исходным кодом.

- Используйте технику разработки через тестирование (test driven development - TDD). Другого пути в действительности не существует! В версии 2 системы Mailman очень не хватало набора автоматизированных тестов и, хотя и справедливо утверждение о том, что не вся кодовая база версии 3 системы Mailman затрагивается существующим набором тестов, большая часть все же тестируется и весь новый код должен быть подвергнут тестированию либо с помощью `unittests`, либо с помощью `doctests`. Использование техники разработки через тестирование является единственным способом проверки того, что все сделанные вами сегодня изменения не приводят к регрессиям в существующем коде. Да, техника разработки через тестирование иногда замедляет процесс разработки, но стоит рассматривать это замедление как инвестицию в будущее качество вашего кода. Таким образом, *отсутствие* качественного набора тестов означает то, что вы просто теряете свое время. Запомните мантру: не протестированный код является неработоспособным.
- Следите за манипуляциями с байтами/строками с самого начала. В Python 3 проведено жесткое разделение между текстовыми строками в кодировке Unicode и байтовыми массивами, которое, хотя изначально и воспринимается болезненно, но впоследствии оказывается значительным преимуществом, используемым для разработки корректного кода. В Python 2 эта граница была размытой, так как строки могли быть как в кодировке Unicode, так и 8-битными ASCII-строками, при этом для их преобразований использовались некоторые автоматизированные механизмы. Несмотря на то, что такое представление строк выглядело достаточно удобным, проблема этой размытой границы была первостепенной причиной ошибок в версии 2 системы Mailman. Также этот способ представления строк не являлся полезным ввиду того факта, что сообщение электронной почты, очевидно, сложно систематизировать как набор строк и байт. Технически представление входящего сообщения электронной почты является байтовой последовательностью, но эти байты практически всегда являются символами из таблицы ASCII и существует большое желание производить манипуляции с сообщением в текстовой форме. Стандарты, относящиеся к сообщениям электронной почты, описывают то, как читаемый человеком текст в отличной от ASCII кодировке может быть безопасно закодирован, поэтому даже такие задачи, как поиск префикса `Re:` в заголовке `Subject:` будет реализован с помощью операций для работы с текстом, а не операций для работы с байтами. Принципом функционирования системы Mailman является преобразование всех входящих данных из байтового представления в строки Unicode так быстро, как это возможно, внутренняя обработка строк с использованием кодировки Unicode и преобразование назад в байтовый поток только при формировании исходящих сообщений. Важно, чтобы были абсолютно ясны с самого начала периоды осуществления обработки байт и периоды осуществления обработки текста, так как это модель будет очень сложно модифицировать впоследствии.

- С самого начала производите интернационализацию вашего приложения. Вам действительно хочется, чтобы ваше приложение использовала только малая часть англоязычных пользователей со всего мира? Подумайте о том, какое количество замечательных пользователей игнорируется! Не так сложно начать процесс интернационализации, причем существует множество хороших инструментов для упрощения этого процесса, многие из которых были впервые применены в системе Mailman. Не беспокойтесь о том, с каких переводов начать; если ваше приложение будет доступно для носителей множества языков со всего мира, в вашу дверь обязательно поступят переводчики с предложениями помочи.

GNU Mailman является развивающимся проектом со здоровой пользовательской базой и огромным количеством возможностей для участия в проекте. Ниже представлены ресурсы, которые вы можете использовать в том случае, если вам захочется помочь нам с разработкой и я надеюсь, что именно это вы и сделаете!

- [Основной вебсайт](#)
- [Ресурс wiki проекта](#)
- [Список рассылки разработчиков](#)
- [Список рассылки пользователей](#)
- Канал IRC в сети Freenode: #mailman

Заключительные слова

Во время написания этой главы мы с глубокой скорбью узнали о кончине Tokio Kikuchi (<http://wiki.list.org/display/COM/TokioKikuchi>), профессора из Японии, который внес большой вклад в разработку Mailman и обладал исключительными знаниями в области вопросов интернационализации и характеристик японских программ для чтения сообщений электронной почты. Нам будет очень не хватать его.

11. Библиотека matplotlib

Глава 11 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

matplotlib является библиотекой для построения графиков, созданной с использованием языка программирования Python и обладающей полной поддержкой графических 2D-операций и ограниченной поддержкой 3D-операций, которая широко используется сформированным сообществом Python-разработчиков, занимающихся научными вычислениями. Библиотека предназначена для применения в широком спектре условий. С помощью нее возможно осуществить встраивание графиков в состав пользовательского интерфейса, созданного с помощью выбранного вами тулкита, при этом на сегодняшний день библиотека поддерживает возможность интерактивного создания графиков в большинстве основных операционных систем, использующих тулкиты GTK+, Qt, Tk, FLTK, wxWidgets и Cocoa. Функции библиотеки могут вызываться интерактивно с помощью командной оболочки языка Python для построения графиков с использованием простых команд процедурного типа, очень похожих на те, что используются в таких системах, как Mathematica, IDL или MATLAB. Библиотека matplotlib также может быть встроена в состав сценария, исполняемого на веб-сервере без оборудования для вывода графики с целью создания графиков в виде файлов как растровых форматов, таких, как Portable Network Graphics (PNG), так и векторных форматов, таких, как PostScript, Portable Document Format (PDF) и Scalable Vector Graphics (SVG), которые замечательно выглядят на бумаге.

11.1. Проблема ключа аппаратной защиты

История создания библиотеки matplotlib берет свое начало с попытки одного из нас (John Hunter) избавить себя и своих исследующих эпилепсию коллег от использования проприетарного программного пакета, предназначенного для анализа электроэнцефалограмм (EcoG). Лаборатория, в которой он работал, обладала только одной лицензией на использование программного обеспечения, поэтому различные выпускники и студенты медицинских университетов, научные сотрудники с учеными степенями, интерны и исследователи по очереди использовали ключ аппаратной за-

щиты программного обеспечения. Система MATLAB широко использовалась в рамках сообщества биомедиков для анализа и визуализации данных, поэтому John Hunter с некоторым успехом смог заменить проприетарное программное обеспечение на версию программного обеспечения на основе системы MATLAB, которая должна была впоследствии использоваться и совершенствоваться многими исследователями. Однако, система MATLAB рассматривает мир как массив чисел с плавающей точкой и сложность реальных медицинских записей состояния больных эпилепсией пациентов с множеством данных условий (CT, MRI, ECoG, EEG) обусловила хранение данных на различных серверах ввиду исчерпания возможностей системы MATLAB в качестве системы управления данными. Убедившись в непригодности системы MATLAB для выполнения поставленной задачи, John Hunter начал работу над новым приложением на языке Python с пользовательским интерфейсом на основе тулкита GTK+, на основе которого также была построена ведущая на тот момент оконная система для Linux.

Таким образом, библиотека matplotlib изначально разрабатывалась как инструмент для визуализации данных EEG/ECoG для этого приложения на основе GTK+, а условия ее использования были продиктованы оригинальной архитектурой. Изначально библиотека matplotlib была спроектирована также с другой целью: выступить заменой инструмента для интерактивной генерации графиков на основе команд, что системе MATLAB удавалось очень хорошо. Архитектура системы MATLAB позволяла выполнять простую задачу загрузки данных из файла и построения на основе этих данных графика достаточно прозрачно, при этом использование полностью объектно-ориентированного API в данном случае привело бы к значительному усложнению синтаксиса. Поэтому библиотека matplotlib также предоставляет не зависящий от состояния интерфейс сценариев для быстрой и простой генерации графиков, аналогично тому, как это реализуется в системе MATLAB. Так как matplotlib является библиотекой, пользователи имеют доступ ко всему множеству встроенных структур данных языка Python, таких, как списки, словари, множества и другим.

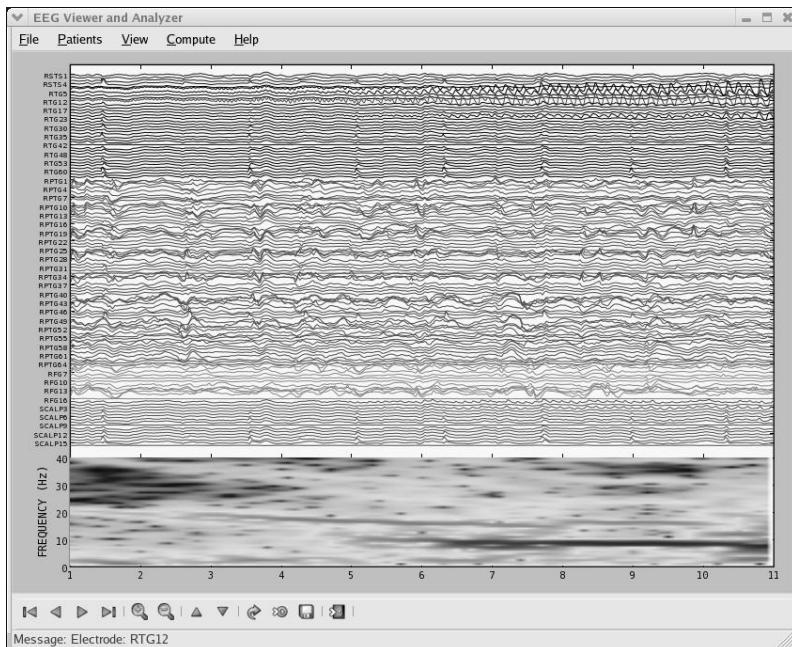


Рисунок 11.1: Оригинальное использующее matplotlib приложение: инструмент для визуализации данных ECoG

11.2. Обзор архитектуры библиотеки matplotlib

Объект библиотеки, находящийся на верхнем уровне и содержащий и управляющий всеми элементами заданного графика, носит имя `Figure`. Одной из основных архитектурных задач, которую должна решать библиотека matplotlib, является реализация фреймворка для представления и манипуляции объектом `Figure` независимо от операции отображения объекта `Figure` в окне пользовательского интерфейса или файле. Это позволяет нам интегрировать усложненные функции и ло-

гические операции в объекты `Figure`, делая системы поддержки вывода данных или системы поддержки устройств вывода относительно простыми. Библиотека `matplotlib` реализует не только интерфейсы рисования для предоставления возможности вывода данных на множество устройств, но также базовые функции обработки событий и создания окон для большинства популярных тулитов, используемых для построения пользовательских интерфейсов. Благодаря этому пользователи могут создавать довольно сложные интерактивные графики и тулиты, поддерживающие ввод с использованием клавиатуры и мыши, которые могут быть подключены без модификаций к шести тулитам для создания пользовательских интерфейсов, поддерживаемым нами.

Архитектура для реализации этих возможностей логически разделена на три уровня, которые могут рассматриваться в виде стека. При этом каждый уровень, находящийся над другим уровнем, знает метод обращения к нижележащему уровню, но нижележащий уровень не располагает информацией об уровнях, находящихся выше него. Три уровня снизу вверх: уровень вывода данных, уровень рисования, уровень сценариев.

Уровень вывода данных

Снизу стека расположен уровень вывода данных (`backend layer`), который предоставляет реализации абстрактных классов интерфейса:

- `FigureCanvas` реализует концепцию поверхности для рисования (т.е., является "бумагой").
- `Renderer` осуществляет рисование (т.е., является "кистью").
- `Event` обрабатывает пользовательский ввод, представленный событиями от таких устройств, как клавиатура и мышь.

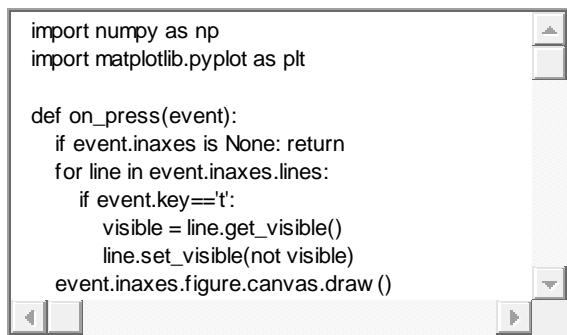
Для такого тулита пользовательского интерфейса, как `Qt`, класс `FigureCanvas` содержит реализацию алгоритма, позволяющего произвести встраивание в созданное с помощью `Qt` окно (`QtGui.QMainWindow`), передать команды класса `Renderer` библиотеки `matplotlib` классу канвы (`QtGui.QPainter`) и преобразовать события тулита `Qt` в события класса `Event` библиотеки `matplotlib`, который отправляет сигналы обработчику обратных вызовов функций для генерации событий, чтобы высокоуровневые обработчики смогли использовать их. Базовые абстрактные классы располагаются в модуле `matplotlib.backend_bases`, а все дочерние классы расположены в таких отдельных модулях, как `matplotlib.backends.backend_qt4agg`. В случае систем вывода данных в файлы изображений, таких, как `PDF`, `PNG`, `SVG` или `PS`, реализация класса `FigureCanvas` может просто инициализировать объект типа файла с описаниями стандартных заголовков, шрифтов и макро-функций наряду с отдельными объектами (линиями, текстом, прямоугольниками, и.т.д.), создаваемыми с помощью класса `Renderer`.

Задачей класса `Renderer` является предоставление низкоуровневого интерфейса рисования для изображения фигур на канве. Как было сказано выше, оригинальное приложение на основе библиотеки `matplotlib` являлось инструментом для визуализации данных `ECOG` на основе тулита `GTK+` и большая часть оригинальной архитектуры была создана под влиянием API `GDK/GTK+`, доступного в тот момент. Оригинальный API класса `Renderer` был создан на основе интерфейса `Drawable GDK`, который реализует такие примитивные методы, как `draw_point`, `draw_line`, `draw_rectangle`, `draw_image`, `draw_polygon` и `draw_glyphs`. В каждой из разрабатываемых нами систем вывода данных - первыми были системы вывода данных в файлы формата `PostScript` и с помощью библиотеки `GD` - был реализован API `GDK Drawable`, после чего его методы преобразовывались в используемые данной системой команды рисования. Как мы обсудили выше, эта необоснованно запутанная реализация новых систем вывода данных с большим количеством методов, а также этот API впоследствии были значительно упрощены, что привело к упрощению процесса портирования библиотеки `matplotlib` для использования новых тулитов пользовательского интерфейса или спецификаций файлов.

Одним из удачно функционирующих архитектурных решений в рамках библиотеки `matplotlib` является поддержка низкоуровневой библиотеки вывода, использующей библиотеку шаблонов язы-

ка C++ с названием Anti-Grain Geometry или "agg" [She06]. Это высокопроизводительная библиотека для вывода 2D-графики со сглаживанием (anti-aliasing), которая позволяет создавать привлекательные изображения. Библиотека matplotlib поддерживает вставку буферов пикселей, выводимых библиотекой agg в качестве элемента пользовательского интерфейса на основе каждого из поддерживаемых нами тулkitов, поэтому становится возможным вывод идентичных с точностью до пикселя графиков в различных пользовательских интерфейсах и операционных системах. Так как при выводе с помощью matplotlib изображений в формате PNG также используется библиотека agg, изображение из файла будет идентичным изображению на экране, поэтому вы увидите график, который не изменится в различных пользовательских интерфейсах, операционных системах и файлах формата PNG.

Фреймворк Event из состава matplotlib связывает такие события уровня пользовательского интерфейса, как key-press-event или mouse-motion-event с классами KeyEvent или MouseEvent из состава matplotlib. Пользователи могут соединить эти события с функциями обратного вызова и осуществлять взаимодействие с их графиками и данными; например, для захвата элемента или множества элементов набора данных (pick) или манипуляции каким-либо аспектом отображения графика или его составных частей. Следующий пример кода иллюстрирует метод переключения отображения всех линий в использующем класс Axes окне при нажатии пользователем клавиши 't'.



Абстракция над фреймворком событий уровня тулкита для создания пользовательского интерфейса позволяет и разработчикам библиотеки matplotlib, и конечным пользователям осуществлять обработку событий пользовательского интерфейса в соответствии с подходом "разработать один раз и использовать везде". Например, функции интерактивной фиксации и масштабирования созданных с использованием библиотеки matplotlib изображений, работающие со всеми тулkitами пользовательского интерфейса, реализуются в рамках фреймворка обработки событий библиотеки matplotlib.

Уровень рисования

Иерархия классов уровня рисования (Artist hierarchy) находится на среднем уровне стека, а также является местом, где производится большая часть сложных операций. Продолжая аналогию, по которой класс FigureCanvas системы вывода данных является бумагой, класс Artist является объектом, который знает, как использовать класс Renderer (кисть) и поместить краску на канву. Все рисунки, принадлежащие классу Figure, которые вы видите, состоят из экземпляров класса Artist; заголовок, линии, метки на осях, изображения и другие элементы соответствуют отдельным экземплярам класса Artist (обратитесь к [Рисунку 11.3](#)). Базовым классом является класс matplotlib.artist.Artist, который содержит атрибуты, свойственные каждому классу Artist: данные преобразования, используемые для преобразования координат класса рисования в координаты канвы (этот процесс описан ниже в подробностях), настройки видимости, координаты области, устанавливающие регион, в котором может осуществляться рисование, строку с названием и интерфейс для осуществления взаимодействия с пользователем, например, для "выбора точек"; это взаимодействие осуществляется путем установления факта нажатия кнопки мыши в области рисования.

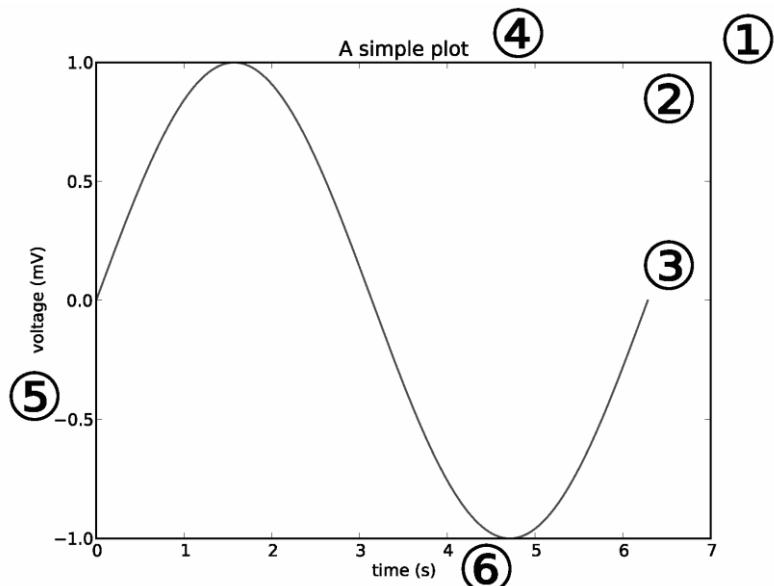
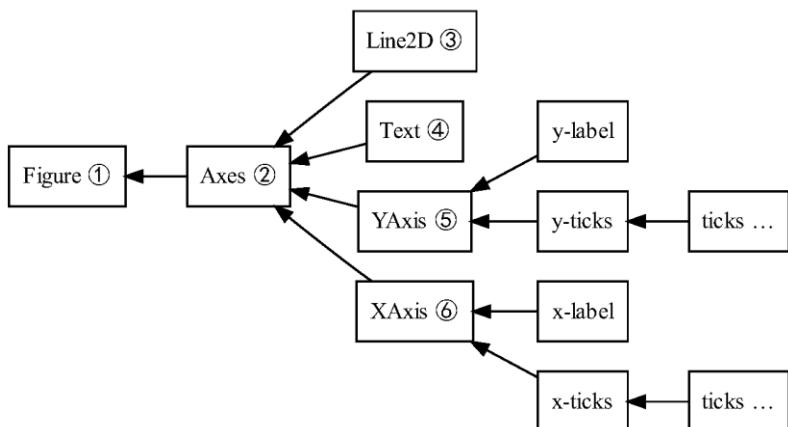


Рисунок 11.2: Готовый рисунок

Рисунок 11.3: Иерархия экземпляров класса уровня рисования, используемая при создании [Рисунка 11.2](#).

Объединение иерархии классов уровня рисования (*Artist hierarchy*) и системы вывода данных осуществляется в рамках метода `draw`. Например, в шаблоне класса, приведенном ниже, где мы создали класс `SomeArtist`, являющийся подклассом класса `Artist`, основным методом, который должен был быть реализован в рамках класса `SomeArtist`, является метод `draw`, с помощью которого примитив для рисования передается от системы вывода данных. Класс `Artist` не располагает информацией о том, какая система вывода данных будет использоваться для рисования (PDF, SVG, GTK+ DrawingArea, и.т.д.), но при этом он располагает информацией о том, как работать с API класса `Renderer` и будет использовать подходящий метод (`draw_text` или `draw_path`). Так как класс `Renderer` содержит указатель на канву и располагает информацией о том, как рисовать на ней, метод `draw` осуществляет преобразование абстрактного представления иерархии классов `Artist` в цвета буфера пикселей, координаты направлений в файле SVG или любое другое конкретное представление.

```
class SomeArtist(Artist):
    'Пример подкласса класса Artist, реализующего'

    def draw(self, renderer):
        """Вызов подходящих методов системы вы
        if not self.get_visible(): return

        # создание некоторых объектов и системы
        renderer.draw_path(graphics_context, path, tr,
```

Существует два типа классов `Artist` в рамках иерархии классов уровня рисования. Примитивные классы (*Primitive artists*) представляют типы объектов, которые вы видите на графике: `Line2D` (двумерная линия), `Rectangle` (прямоугольник), `Circle` (окружность) и `Text` (текст). Композитные объекты (*Composite artists*) являются наборами классов `Artist`, такими, как классы `Axis` (ось), `Tick` (метки), `Axes` (координатные оси) и `Figure` (рисунок). Каждый композитный класс может содержать другие композитные классы также, как и примитивные классы. Например, класс `Figure` содержит один или несколько композитных классов `Axes` и фон изображения, представленного классом `Figure`, создан с помощью примитивного класса `Rectangle`.

Наиболее важным композитным классом является класс `Axes`, в рамках которого объявлена большая часть методов для создания графиков из состава API библиотеки `matplotlib`. Класс `Axes` содержит не только большую часть графических элементов, формирующих фон графика - метки, линии координатные оси, сетку, цвет, используемый для фона графика, но и множество вспомогательных методов для создания примитивных классов и добавления их в экземпляр класса `Axes`. Например, в [Таблице 11.1](#) показан пример нескольких методов класса `Axes`, с помощью которых объекты графика создаются и сохраняются в экземпляре класса `Axes`.

Таблица 11.1: Пример методов класса `Axes` и экземпляров класса `Artist`, которые создаются с помощью них

Метод	Создает класс	Хранится в
<code>Axes.imshow</code>	Один или несколько экземпляров <code>matplotlib.image.AxesImage</code>	<code>Axes.images</code>
<code>Axes.hist</code>	Множество экземпляров <code>matplotlib.patch.Rectangle</code>	<code>Axes.patches</code>
<code>Axes.plot</code>	Один или несколько экземпляров <code>matplotlib.lines.Line2D</code>	<code>Axes.lines</code>

Ниже приведен простой сценарий на языке Python, иллюстрирующий описанные выше архитектурные решения. Он устанавливает систему вывода данных, соединяет с ней экземпляр класса `Figure`, использует библиотеку для работы с массивами `numpy` для генерации 10000 нормально распределенных случайных чисел и формирует их гистограмму.

```
# Импортируется класс FigureCanvas из выбран
# после чего к нему привязывается экземпляр к
from matplotlib.backends.backend_agg import Figure
from matplotlib.figure import Figure
fig = Figure()
canvas = FigureCanvas(fig)

# Импортируется библиотека numpy для генера
import numpy as np
x = np.random.randn(10000)
```

Уровень сценариев (pyplot)

Приведенный выше сценарий, использующий API, работает очень хорошо, особенно в случае его использования разработчиками и примененная программная парадигма подходит для разработки серверных веб-приложений, приложений с пользовательским интерфейсом, или, возможно, для разработки сценариев и обмена ним со сторонними разработчиками. Для каждодневной работы, особенно в случае интерактивной исследовательской работы ставящих опыты ученых, которые не являются профессиональными программистами, синтаксис является в некоторой степени усложненным. Большинство специализированных языков программирования для анализа данных и их визуализации предоставляет простой интерфейс сценариев для упрощения выполнения стандартных задач, при этом библиотека matplotlib предоставляет интерфейс `matplotlib.pyplot` для выполнения аналогичных действий. Код, представленный выше, будет выглядеть следующим образом в случае использования `pyplot`:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(10000)
plt.hist(x, 100)
plt.title(r'Normal distribution with $\mu=0, \sigma=1$')
plt.savefig('matplotlib_histogram.png')
plt.show()
```

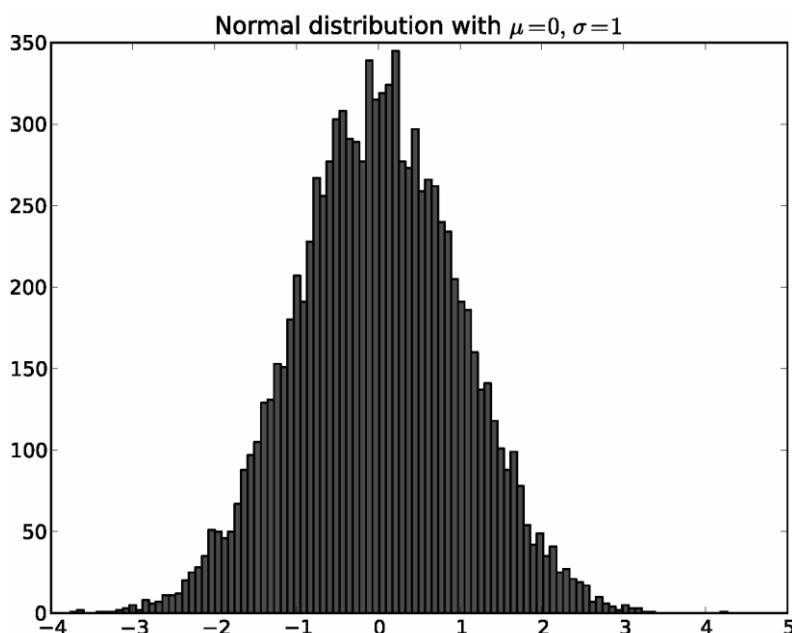


Рисунок 11.4: Гистограмма, созданная с помощью `pyplot`

Интерфейс `pyplot` является интерфейсом, изменяющим параметры состояния и выполняющим команды создания графиков с осями координат с последующим соединением их с выбранной системой вывода данных, а также поддерживающим внутренние структуры данных уровня модулей, представляющие текущий рисунок и координатные оси, в отношении которых непосредственно могут выполняться команды.

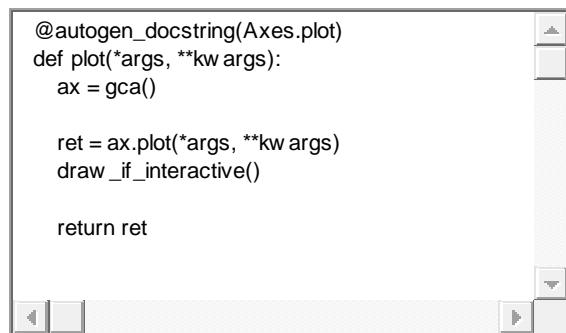
Давайте рассмотрим важные строки сценария для того, чтобы понять принцип управления внутренними данными состояния.

- `import matplotlib.pyplot as plt`: В момент загрузки модуля `pyplot` происходит разбор локального файла конфигурации, в котором помимо различных параметров хранятся пользовательские настройки, содержащие выбранную систему вывода данных. Это может быть система вывода данных в пользовательский интерфейс, такая, как `QtAgg` и в этом случае приведенный выше сценарий импортирует модуль фреймворка для создания графического пользователь-

ского интерфейса и создаст окно Qt с графиком, а также в качестве системы вывода данных может быть выбрана система вывода данных в файл, такая, как Agg и в этом случае сценарий генерирует файл и завершит свою работу.

- `plt.hist(x, 100)`: Это первая команда создания графика в сценарии. Интерфейс `pyplot` проверит внутренние структуры данных для установления того, присутствует ли экземпляр класса `Figure` для данного графика. В случае его наличия, он извлечет текущий экземпляр класса `Axes` и осуществит непосредственное создание графика путем вызова метода API `Axes.hist`. В том случае, если этого экземпляра класса не существует, будут созданы экземпляры классов `Figure` и `Axes`, которые будут сделаны текущими, после чего будет осуществлено непосредственное создание графика при помощи вызова метода `Axes.hist`.
- `plt.title(r'Normal distribution with $\mu=0, \sigma=1$')`: Как было описано выше, интерфейс `pyplot` установит, присутствуют ли текущие экземпляры классов `Figure` и `Axes`. В случае их обнаружения он не будет создавать новых экземпляров, а вместо этого осуществит прямой вызов метода `Axes.set_title` существующего экземпляра класса `Axes`.
- `plt.show()`: Этот вызов приведет к выводу изображения с помощью экземпляра класса `Figure` и в том случае, если пользователь выбрал систему вывода данных, использующую тулкит для создания графического интерфейса в файле конфигурации, будет запущен основной цикл приема событий графического интерфейса и пользователю будут показаны созданные изображения.

Незначительно сокращенная и упрощенная версия часто используемой функции для создания линий `matplotlib.pyplot.plot` интерфейса `pyplot` показана ниже для иллюстрации метода реализации функциональности объектно-ориентированного основного интерфейса `matplotlib` с помощью функции интерфейса `pyplot`. Другие функции интерфейса сценариев `pyplot` используют аналогичные архитектурные решения:



Директива языка Python `@autogen_docstring(Axes.plot)` извлекает строку документации для соответствующего метода API и добавляет ее корректно отформатированную версию в метод `pyplot.plot`; у нас имеется отдельный модуль `matplotlib.docstring` для этой магии со строками документации. Аргументы `*args` и `**kwargs` в документации используют специальные соглашения языка Python для указания всех аргументов и ключевых слов, относящихся к аргументам, предназначенных для передачи в качестве данных метода. Это позволяет нам перенаправить их соответствующему методу из состава API. Вызов `ax = gca()` позволяет использовать механизм изменения параметров состояния для получения "текущих экземпляров класса `Axes`" (каждый интерпретатор языка Python может иметь только один "текущий экземпляр класса `Axes`"), а также позволяет создать экземпляры классов `Figure` и `Axes` в случае необходимости. Вызов `ret = ax.plot(*args, **kwargs)` перенаправляет аргументы соответствующему методу экземпляра класса `Axes` и сохраняет возвращаемое значение для последующего возврата. Таким образом, интерфейс `pyplot` является достаточно тонкой оберткой над API основного класса `Artist`, при создании которой была предпринята попытка избежать копирования кода настолько, насколько это возможно путем раскрытия функций API, спецификаций вызова и строк документации в рамках интерфейса сценариев с минимальным количеством лишнего кода.

11.3. Рефакторинг системы вывода данных

Со временем количество методов API для рисования системы вывода данных неуклонно росло, при этом использовались следующие методы:

`draw_arc, draw_image, draw_line_collection, draw_line, draw_lines, draw_point,`

```
draw_quad_mesh, draw_polygon_collection, draw_polygon, draw_rectangle,
draw_regpoly_collection
```

К сожалению, использование большого количества методов системы вывода данных усложняло разработку новых систем вывода, а так как новые функции добавлялись в основной код, обновление существующих систем вывода данных также стало достаточно сложной задачей. Так как каждая из систем вывода данных была реализована силами одного разработчика, который являлся экспертом в области определенного формата файлов, иногда требовалось большое количество времени для реализации новой возможности в каждой из систем вывода данных, что еще больше затрудняло понимание пользователем того, какие функции доступны в той или иной системе.

В версии 0.98 библиотеки matplotlib код систем вывода данных подвергся рефакторингу с целью переноса всех функций из систем вывода данных, за исключением необходимых, в основной код библиотеки таким образом, чтобы в составе систем вывода данных остался минимум необходимой функциональности. Количество требуемых методов в рамках API систем вывода данных было значительно сокращено до следующих методов:

- `draw_path`: Рисует сложные полигоны с помощью линий и сегментов Безье. Этот интерфейс заменяет множество устаревших методов: `draw_arc`, `draw_line`, `draw_lines` и `draw_rectangle`.
- `draw_image`: Выводит растровые изображения.
- `draw_text`: Выводит текст с заданными параметрами шрифта.
- `get_text_width_height_descent`: При передаче строки текста возвращает ее метрики.

Возможно реализовать все необходимые функции рисования в рамках новой системы вывода данных, используя только перечисленные методы. (Мы можем пойти еще дальше и выводить текст с помощью метода `draw_path`, исключив тем самым необходимость в реализации метода `draw_text`, но мы не захотели реализовывать данное упрощение. Конечно же, в рамках системы вывода данных возможна реализация специфического метода `draw_text` для вывода "реального" текста.) Эти изменения упростили процесс создания и разработки новой системы вывода. Однако, в некоторых случаях система вывода данных может потребоваться изменить принцип работы основного кода библиотеки для увеличения производительности операции вывода. Например, при рисовании маркеров (небольших символов, используемых для указания на вершины графика) для большей экономии места удобно однократно записывать изображение маркера в файл, после чего использовать его в необходимых местах, копируя методом "штампа". В этом случае система вывода данных может реализовать метод `draw_markers`. Если этот метод реализован, система вывода данных будет однократно записывать изображение маркера, после чего будет отправлять гораздо более короткую команду для повторного использования этого изображения во множестве точек. Если же этот метод не реализован, основной код библиотеки просто будет многократно выводить изображение маркера с помощью метода `draw_path`.

Полный список дополнительных методов API системы вывода данных:

- `draw_markers`: Изображает набор маркеров.
- `draw_paths_collection`: Изображает набор контуров.
- `draw_quad_mesh`: Изображает четырехугольную сетку.

11.4. Преобразования

Библиотека matplotlib тратит большое количество времени на операции преобразования координат из одной системы в другую. Во множество этих системы координат входят:

- **Данные (data)**: не преобразованные значения из данных
- **Оси (axes)**: пространство, заданное определенными координатными плоскостями
- **Рисунок (figure)**: пространство, содержащее весь рисунок
- **Отображение (display)**: физические координаты, использованные для вывода (т.е., точки в PostScript, пиксели в PNG)

Каждый класс `Artist` имеет узел преобразования, содержащий данные о том, как произвести преобразование из одной системы координат в другую. Эти узлы преобразования объединены друг с другом в рамках ориентированного графа, в котором каждый узел зависит от родительского узла. По пути от ребер к корню графа, координаты пространства данных для любой вершины могут быть преобразованы в координаты результирующего файла. Большинство преобразований также является обратимым. Это обстоятельство позволяет выбрать элемент графика и получить его координаты пространства данных. Граф преобразований устанавливает зависимости между вершинами: при изменении данных преобразования для корня графа, таком, как изменение границ координатных осей в классе `Axes`, все данные преобразований, связанные с классом `Axes` становятся недействительными, так как точки должны быть перерисованы. Данные преобразований, связанные с другими классами `Axes` для рисунка, конечно же, не должны затрагиваться с целью предотвращения ненужных повторных расчетов и повышения интерактивности операций.

Узлы преобразований могут выполнять как простые афинные, так и неафинные преобразования. Афинные преобразования являются семейством преобразований, сохраняющих прямыми линии и соотношения расстояний изображения, выполняя его вращение, преобразование, масштабирование и наклон. Двумерные афинные преобразования представляются с помощью матрицы афинного преобразования размерностью 3x3. Координаты точки после преобразования (x', y') вычисляются путем умножения матрицы с начальными координатами (x, y) на следующую матрицу:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & \theta_x & t_x \\ \theta_y & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Координаты двумерного пространства могут быть просто преобразованы путем умножения их на матрицу трансформации. Афинные трансформации также обладают полезным свойством, заключающимся в том, что они могут быть объединены с помощью матричного умножения. Это значит, что для выполнения серий афинных преобразований матрицы трансформации могут быть перемножены только один раз, после чего результирующая матрица может быть использована для преобразования координат. Фреймворк преобразования координат библиотеки `matplotlib` автоматически объединяет (замораживает) матрицы афинных преобразований для сокращения объема вычислений. Возможность использования быстрых афинных преобразований важна, так как с помощью них можно повысить производительность интерактивного перемещения и масштабирования изображения в окне графического интерфейса приложения.

Не являющиеся афинными преобразования в рамках библиотеки `matplotlib` используют функции языка Python, поэтому они являются действительно произвольными. В рамках основного кода библиотеки `matplotlib` неафинные преобразования используются для логарифмического масштабирования, создания графиков в полярных координатах и создания географических проекций ([Рисунок 11.5](#)). Эти неафинные преобразования могут свободно смешиваться с афинными в графике преобразования. Библиотека `matplotlib` автоматически упростит афинные преобразования и перейдет к использованию произвольных функций только для части неафинных преобразований.

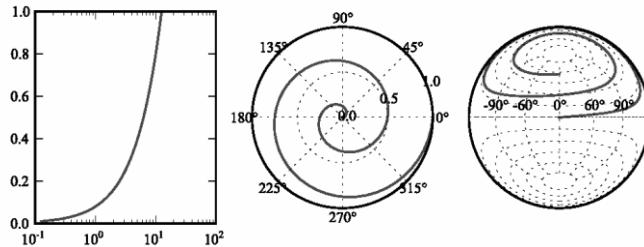


Рисунок 11.5: Одни и те же данные на графиках, подвергнутые трем неафинным преобразованиям: к логарифмическим координатам, к полярным координатам и к координатам проекции Ламберта

Используя эти простые операции, библиотека matplotlib может выполнять некоторые достаточно сложные задачи. Смешанное преобразование выполняется специальным узлом преобразования и предназначено для выполнения одного преобразования для оси x и другого преобразования для оси y . Это, конечно же, становится возможным только в случае, если рассматриваемые преобразования являются "разделяемыми", что подразумевает независимость координат x и y , ну а сами преобразования могут быть как афинными, так и неафинными. Эта возможность используется, например, для создания графиков в логарифмической системе координат, где одна или обе оси x и y могут использовать логарифмическую систему координат. Возможность использования смешанного преобразования позволяет совмещать доступные системы координат произвольным образом. Другой возможностью графа преобразования является разделение осей. Существует возможность "связать" ограничения одного графика с другим и быть уверенным в том, что при перемещении или масштабировании одного из графиков, состояние другого графика будет изменено соответствующим образом. В этом случае один и тот же узел преобразования просто совместно используется двумя осями, которые могут относиться даже к разным изображениям. На [Рисунке 11.6](#) показан пример графа преобразования с задействованием некоторых из этих дополнительных возможностей. Ось $axes1$ является осью x в логарифмической системе координат; оси $axis1$ и $axis2$ совместно используют одну и ту же ось y .

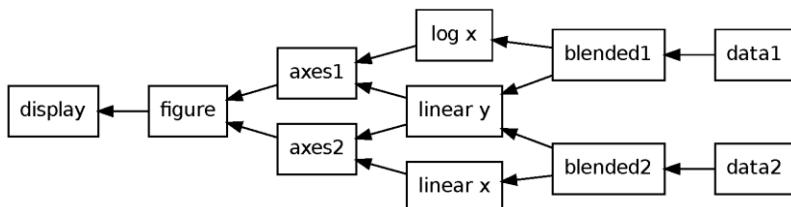


Рисунок 11.6: Пример графа преобразования

11.5. Процесс обработки полилиний

При создании графика с помощью линий библиотека matplotlib выполняет ряд шагов для преобразования необработанных данных в линию на экране. В ранних версиях matplotlib эти шаги были взаимосвязаны. С тех пор код подвергся рефакторингу, поэтому они стали отдельными шагами процесса "преобразования путей". Это обстоятельство позволяет каждой системе вывода данных выбирать шаги процесса обработки для непосредственного выполнения, так как некоторые шаги полезны только в определенных контекстах.

- Преобразование:** Координаты преобразуются из координат данных в координаты изображения. Если это преобразование является полностью афинным, как описано выше, этот процесс заключается в простом умножении матриц. Если требуются произвольные преобразования, вызываются функции для преобразования координат в пространство изображения.
- Обработка отсутствующих данных:** Массив данных может содержать участки, данные из которых отсутствуют или являются некорректными. Пользователь может указать на это либо приравняв их значения к `NaN`, либо использовав маскированные массивы библиотеки `numpy`. Векторные форматы файлов, такие, как PDF и библиотеки вывода изображений, такие, как `Agg` обычно не используют концепцию отсутствующих данных при построении полилиний, поэтому данный шаг процесса должен пропускать сегменты отсутствующих данных с помощью команд `MOVE TO`, которые указывают библиотеке вывода изображения на необходимость перемещения начальной точки перед продолжением рисования.
- Вырезка:** Наличие точек вне границ изображения может привести к увеличению размера файла из-за включения множества невидимых точек. Еще более важно то, что очень большие или очень малые значения координат могут привести к

ошибкам переполнения при создании выходного файла, которые в результате приведут к полностью испорченному выходному файлу. На этом шаге процесса осуществляется обрезка изображения на основе точек входа и выхода полилиний за границы изображения для преодоления всех этих проблем.

- **Привязка к центру пикселей:** идеально вертикальные и горизонтальные линии могут выглядеть размыто из-за сглаживания в том случае, если их центры не выровнены по центру пикселя (обратитесь к [Рисунку 11.7](#)). При выполнении шага привязки к центру пикселей в процессе обработки полилиний в первую очередь устанавливается, состоит ли полилиния только из горизонтальных и вертикальных сегментов (как в случае прямоугольника со сторонами, ориентированными по координатным осям), и, в случае установления этого факта, сдвигает каждую вершину результирующей полилинии к ближайшему центру пикселя. Этот шаг выполняется только для систем вывода данных в растровые форматы, так как векторные форматы должны продолжить использовать точные данные. Некоторые программы просмотра файлов векторных форматов, такие, как Adobe Acrobat, выполняют операции привязки к центру пикселей при просмотре на экране.



Рисунок 11.7: Операция привязки к центру пикселей крупным планом: слева линия без привязки; справа - с привязкой.

- **Упрощение:** При построении действительно сложных графиков множество точек на линии может быть на самом деле невидимым. Это особенно актуально в случае отображения амплитуды шума аудиосигнала на графике. Включение этих точек в график увеличивает размер файла и даже может привести к достижению ограничения разрешенного количества точек для данного формата. Следовательно, любые точки, лежащие непосредственно на линии между двумя соседними точками, удаляются (обратитесь к [Рисунку 11.8](#)). Алгоритм определения использует пороговое значение, устанавливаемое на основе того, что будет видимо при заданном пользователем разрешении.

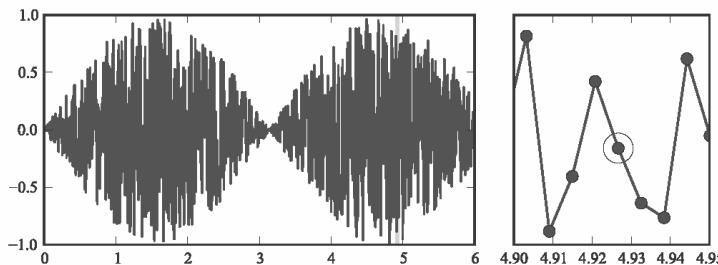


Рисунок 11.8: Изображение справа является увеличенным фрагментом изображения слева. Обведенная вершина автоматически удаляется с помощью механизма упрощения, так как она лежит непосредственно на линии между соседними вершинами и, следовательно, является избыточной.

11.6. Математические выражения

Так как пользователями библиотеки `matplotlib` обычно являются ученые, очень полезно иметь возможность вывода отформатированного текста, содержащего математические выражения, прямо на графике. Скорее всего, самым широко используемым синтаксисом для формирования математических выражений является синтаксис, используемый созданной Donald Knuth системой `TeX`. Он позволяет использовать входные данные в форме обычной текстовой строки, подобной следующей:

```
\sqrt{\frac{\delta x}{\delta y}}
```

и формировать на основе нее отформатированное математическое выражение.

Библиотека `matplotlib` предоставляет два варианта вывода математических выражений. Первый вариант, `usetex`, использует полную копию системы `TeX` на пользовательской машине для вывода математического выражения. Система `TeX` выводит данные о расположении символов и линий для формирования выражения в используемом формате `DVI` (независимо от устройства). После этого `matplotlib` разбирает этот файл `DVI` и преобразует его в набор команд рисования, с помощью которых одна из систем вывода данных сможет нанести выражение непосредственно на график. Этот подход позволяет обрабатывать самый запутанный синтаксис математических выражений. Однако, он требует от пользователя наличия полной установленной рабочей копии системы `TeX`. Поэтому библиотека содержит также внутреннюю систему вывода математических выражений, называемую `mathtext`.

Система `mathtext` является прямым портом системы вывода математических выражений из TeX, объединенным с более простой системой разбора текста, разработанной с использованием фреймворка для разбора текста `pyParsing` [McG07]. Этот порт был создан на основе опубликованной копии исходного кода TeX [Knu86]. Эта простая система разбора текста создает дерево из контейнеров (*boxes*) и связей (*glue* в терминологии TeX), которое после этого используется системой вывода данных. Хотя полная версия системы вывода математических выражений TeX и включается в комплект поставки, большой набор сторонних математических библиотек TeX и LaTeX из него исключается. Возможности этих библиотек переносятся в случае необходимости, с преимуществом для часто используемых и не являющихся специфичными для какой-либо области науки возможностей. Этот подход позволяет создать замечательный и не ресурсоемкий способ вывода математических выражений.

11.7. Тестирование с целью поиска регрессий

Исторически библиотека `matplotlib` не содержала большого количества низкоуровневых модульных тестов. Время от времени при получении сообщений о серьезной ошибке сценарий для ее воспроизведения добавлялся в специально предназначеннную для таких файлов директорию дерева исходного кода. Отсутствие автоматизированных тестов приводило к обычным для такой ситуации проблемам и, что особенно важно, к регрессиям в ранее работающих функциях. (Нам скорее всего не следует внушать вам идею о том, что автоматизированное тестирование является полезной возможностью.) Конечно же, при наличии такого большого объема кода и множества параметров конфигурации, а также взаимозаменяемых частей кода (т.е., систем вывода данных), становится спорным утверждение о том, что для тестирования будет достаточно исключительно низкоуровневых модульных тестов; мы наоборот считаем, что наиболее эффективным является тестирование всех частей кода, функционирующих взаимосвязанно.

С этой целью был разработан сценарий, генерирующий множество графиков, при построении которых используются различные функции библиотеки `matplotlib`, в особенности те, которые было достаточно сложно реализовать. Этот подход немного облегчил процесс установления того, что новое изменение привело к непреднамеренному нарушению работы функции приложения, но корректность изображений все еще приходилось проверять вручную. Так как данное тестирование требовало большого количества ручной работы, оно не производилось достаточно часто.

На втором этапе этот метод был автоматизирован. Используемый на данный момент сценарий тестирования библиотеки `matplotlib` генерирует множество графиков, но вместо требования ручной обработки, эти графики автоматически сравниваются с образцами изображений. Все тесты используют фреймворк тестирования `nose`, который упрощает генерацию отчетов о непройденных тестах.

Усложняющим работу обстоятельством является тот факт, что сравнение изображений не может быть точным. Незначительные изменения версий библиотеки вывода шрифтов `Freetype` могут быть причиной незначительных отличий в выводе текста на различных машинах. Этих отличий не достаточно для того, чтобы считать график "некорректным", но достаточно, чтобы обнаружить различие в ходе побитового сравнения. Вместо этого фреймворк тестирования создает гистограммы обоих изображений и рассчитывает на их основе среднеквадратичное отклонение. Если это отклонение превышает заданное пороговое значение, считается что на изображениях слишком много различий и тест завершается неудачей. При неудачах в ходе проведения тестов генерируются изображения отличий, которые указывают на то, где произошли изменения графика (обратитесь к [Рисунку 11.9](#)). После этого разработчик может решить, является ли это различие результатом намеренного изменения и обновить образец графика для соответствия новому варианту изображения, или изображение на самом деле не является корректным, что подразумевает поиск и исправление ошибки, вызвавшей изменение.

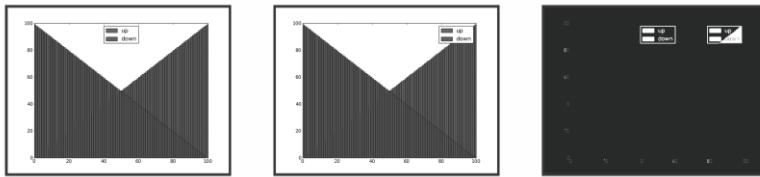


Рисунок 11.9: Сравнение изображений в ходе тестирования с целью поиска регрессий. Слева направо: а) Ожидаемое изображение, б) результат нарушения расположения легенды, в) различие между двумя изображениями.

Так как различные системы вывода данных могут содержать различные ошибки, фреймворк тестирования должен использовать множество систем вывода данных для каждого из графиков: PNG, PDF и SVG. В случае использования векторных форматов мы не сравниваем напрямую информацию из файлов, так как существует множество способов для отображения чего-либо с одинаковым результатом после преобразования в растровый формат. Системы вывода данных в векторный формат должны иметь полную свободу в плане изменения специфики своего вывода с целью повышения производительности без нарушения выполнения всех тестов. Следовательно, в случае систем для вывода данных в векторные форматы фреймворк тестирования в первую очередь преобразует файл в растровый формат с помощью стороннего инструмента (Ghostscript для PDF и Inkscape для SVG), после чего файл в растровом формате используется для сравнения.

Используя этот подход, нам удалось создать достаточно производительный фреймворк тестирования с нуля, причем это оказалось проще, чем разработка множества низкоуровневых модульных тестов. Все же, этот фреймворк не является идеальным; покрытие кода тестами не является полным, а также тратится большое количество времени для выполнения всех тестов. (Около 15 минут на системе с центральным процессором Intel Core 2 E6550 с тактовой частотой 2.23GHz.) Следовательно, некоторые регрессии все еще могут выпадать из области действия тестов, но все же общее качество релизов значительно улучшилось с момента реализации фреймворка тестирования.

11.8. Выученные уроки

Один из наиболее важных уроков, усвоенных в ходе разработки библиотеки matplotlib, может быть описан с помощью выражения Le Corbusier: "Хорошие архитекторы используют заимствования". Авторы ранних версий библиотеки matplotlib были в основном учеными, самостоятельно изучившими программирование и пытавшимися выполнить поставленную задачу, а не обученными специалистами в области компьютерных наук. Поэтому с первого раза и не была создана подходящая внутренняя архитектура библиотеки. Решение о реализации доступного пользователю уровня сценариев, в большей степени совместимого с API MATLAB, принесло пользу проекту в трех различных аспектах: был предоставлен проверенный временем интерфейс для создания и изменения графиков, стал возможным простой переход к использованию matplotlib частью пользователей из большой пользовательской базы системы MATLAB, и, что наиболее важно для нас в контексте архитектуры matplotlib, у разработчиков появилась возможность провести несколько рефакторингов внутреннего объектно-ориентированного API без вмешательства в работу пользователей, так как интерфейс сценариев оставался неизменным. Хотя у нас также были и пользователи API (в отличие от пользователей интерфейса сценариев), большая часть этих пользователей обладала достаточным опытом, чтобы адаптировать свои разработки к изменениям API. Пользователи интерфейса сценариев, с другой стороны, могут разрабатывать код один раз и обладать объективной уверенностью в том, что он будет работать стабильно со всеми последующими релизами.

В ходе реализации внутреннего API рисования, хотя мы и использовали заимствования из GDK, не было приложено достаточных усилий для того, чтобы выяснить, подходит ли в нашем случае данный API рисования, поэтому нам пришлось затратить значительные усилия для того, чтобы после завершения разработки множества систем вывода данных с использованием этого API, расширить

функции этих систем путем использования более простого и гибкого API рисования. Нам было бы удобно использовать спецификацию для операций рисования формата PDF [[Ent11b](#)], которая была создана с учетом многолетнего опыта работников компании Adobe, полученного при создании спецификации формата PostScript; это позволило бы нам получить начальную совместимость с самим форматом PDF, фреймворком Quartz Core Graphics и инструментарием для рисования Enthought Enable Kiva [[Ent11a](#)].

Одним из недостатков языка Python является то, что из-за простоты и выразительности этого языка разработчики обычно считают, что проще повторно спроектировать и реализовать необходимые существующие в других пакетах функции, чем провести работу по интеграции кода из этих пакетов. Для библиотеки matplotlib на ранних этапах развития была бы полезной интеграция с существующими модулями и API, такими, как тулкиты Kiva и Enable от организации Enthought, которые решают аналогичные проблемы, вместо повторной реализации функций. Интеграция с существующими пакетами, однако, является обоюдоострым мечом, так как она может сделать сборки и релизы более сложными и негативно повлиять на гибкость внутренней разработки.

12. MediaWiki

Глава 12 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

С самого начала приложение MediaWiki разрабатывалось как специфический программный продукт для проекта Wikipedia. Деятельность разработчиков была направлена на упрощение возможности его повторного применения сторонними пользователями, но процесс разработки и предназначение для работы в рамках проекта Wikipedia значительным образом повлияли на архитектуру приложения MediaWiki в течение истории его развития.

Wikipedia является одним из десяти наиболее популярных вебсайтов в мире, на данный момент обслуживающим около 400 миллионов уникальных посетителей в месяц. Этот сайт обрабатывает более 100000 обращений в секунду. Вебсайт Wikipedia не использует коммерческие рекламные объявления для финансирования своей деятельности; он поддерживается в полном объеме некоммерческой организацией Wikimedia Foundation, основным источником финансирования которой являются пожертвования. Это значит, что программный продукт MediaWiki должен не только работать на находящемся в первой десятке по популярности сайте, но также делать это в условиях ограниченного бюджета. Для удовлетворения этих требований приложение MediaWiki должно было быть разработано с большим вниманием к производительности, возможностям кэширования и оптимизациям. Ресурсоемкие возможности, которые не могут быть включены в Wikipedia либо удаляются, либо отключаются с помощью переменных конфигурации; в ходе разработки осуществляется бесконечное балансирование между производительностью и возможностями приложения.

Влияние проекта Wikipedia на архитектуру системы MediaWiki не ограничивается производительностью. В отличие от систем управления содержимым вебсайта (CMS) общего назначения, приложение MediaWiki изначально разрабатывалось со специфической целью: для поддержки сообщества, которое создает и курирует свободно распространяемые знания в рамках открытой платформы. Это значит, например, что MediaWiki не поддерживает такие возможности, реализуемые в рамках корпоративных систем управления содержимым вебсайта, как поток обработки публикаций или списки контроля доступа, но предоставляет множество различных инструментов для борьбы со спамом и вандализмом.

Таким образом, с самого начала требования и действия постоянно растущего сообщества участников проекта Wikipedia повлияли на процесс разработки приложения MediaWiki и наоборот. Архитектура приложения MediaWiki множество раз претерпевала изменения благодаря таким начатым или предложенным сообществом инициативам, как создание проекта Wikimedia Commons или

функция флагов для изменений (Flagged Revisions). Разработчики реализовывали основные архитектурные изменения при появлении их необходимости в процессе использования приложения MediaWiki участниками проекта Wikipedia.

Приложение MediaWiki также приобрело обширный круг сторонних пользователей благодаря изначальному развитию в форме программного обеспечения с открытым исходным кодом. Сторонние пользователи знают о том, что пока такой популярный вебсайт, как Wikipedia, использует приложение MediaWiki, это программное обеспечение будет поддерживаться в рабочем состоянии и совершенствоваться. Приложение MediaWiki на самом деле в первую очередь предназначалось для сайтов организации Wikimedia, но были предприняты усилия для того, чтобы расширить область его применения и сделать его более подходящим для удовлетворения потребностей сторонних пользователей. Например, в данный момент в составе приложения MediaWiki поставляется замечательный веб-установщик, упрощающий процесс установки по сравнению с процессом, в ходе которого все действия должны выполняться с помощью командной оболочки и программное обеспечение содержит жестко заданные пути к файлам, специфичные для проекта Wikipedia.

Все же, приложение MediaWiki остается программным обеспечением проекта Wikipedia, что видно при рассмотрении его истории развития и архитектуры.

Данная глава организована следующим образом:

- Раздел "[Исторический обзор](#)" содержит краткое описание истории приложения MediaWiki или, точнее, его предыстории и условий создания.
- Раздел "[Кодовая база и практика разработки приложения MediaWiki](#)" объясняет причины выбора языка PHP, важность и принципы создания безопасного кода, а также то, как обрабатываются стандартные настройки.
- Раздел "[База данных и хранилище текста](#)" подробно описывает систему распределенного хранения данных, а также то, как структура этой системы адаптируется к росту объема данных.
- Раздел "[Запросы, кэширование и доставка данных](#)" прослеживает выполнение веб-запроса в тесной связи с активируемыми компонентами системы MediaWiki. Раздел также включает в себя описание различных уровней кэширования и системы доставки данных.
- Раздел "[Языки](#)" в подробностях описывает всеобъемлющую систему интернационализации и локализации, а также причины, по которым она нужна и способ ее реализации.
- Раздел "[Пользователи](#)" содержит описание способа программного представления пользователей и принципа функционирования пользовательских прав доступа.
- Раздел "[Содержимое страниц](#)" подробно описывает способы структурирования содержимого, форматируемого и обрабатываемого для генерации конечного документа HTML. Подраздел посвящен обработке мультимедийных файлов средствами приложения MediaWiki.
- Раздел "[Модификации и расширение возможностей MediaWiki](#)" описывает то, как сценарии JavaScript, таблицы стилей CSS, расширения и оболочки могут использоваться для модификации wiki, а также как они модифицируют внешний вид и поведение системы. Подраздел посвящен веб-API с возможностью программного чтения данных.

12.1. Исторический обзор

Фаза I: UseModWiki

Проект Wikipedia был начат в январе 2001 года. В этот момент он был в большей степени экспериментом, проводимым с целью проверки возможности ускоренной генерации содержимого для проекта Numpedia, энциклопедии с бесплатными, но оцениваемыми пользователями данными, созданной Jimmy Wales. Так как проект был экспериментом, энциклопедия Wikipedia изначально работала под управлением UseModWiki, существующей в тот момент системы wiki, разработанной с использованием языка Perl и компонента CamelCase, хранящей все страницы в виде отдельных текстовых файлов без записи истории сделанных изменений и распространявшейся в соответствии с условиями лицензии GPL.

Позднее стало ясно, что компонент CamelCase не подходит для именования статей энциклопедии. В конце января 2001 года разработчик UseModWiki и участник проекта Wikipedia Clifford Adams добавил новую возможность в UseModWiki: свободные ссылки; т.е., возможность связывания страниц с использованием специальных синтаксических конструкций (двойных квадратных скобок).

бок) вместо автоматических ссылок, создаваемых CamelCase. Спустя несколько недель, проект Wikipedia перешел к использованию новой версии системы UseModWiki с поддержкой свободных ссылок и включил эту возможность.

Хотя эта начальная фаза развития и не относится к процессу создания системы MediaWiki, она описывает контекст развития и показывает, что даже перед созданием приложения MediaWiki, проект Wikipedia развивал возможности используемого программного обеспечения в соответствии со своими требованиями. Приложение UseModWiki также повлияло на некоторые возможности приложения MediaWiki; например, на его язык разметки. Страница [ностальгии проекта Wikipedia](#) содержит полную копию базы данных Wikipedia на декабрь 2001 года, когда проект Wikipedia все еще использовал приложение UseMediaWiki.

Фаза II: Сценарий PHP

В 2001 году сайт Wikipedia не находился в первой десятке популярных вебсайтов; это был малоизвестный проект, расположенный в неизвестной области сети, не посещаемый большинством поисковых машин и работающий на единственном сервере. При этом производительность уже была проблемой, в первую очередь из-за того, что приложение UseModWiki хранило свои данные в базе данных, представленной в виде обычных файлов. В это время участники проекта Wikipedia опасались того, что вебсайт не выдержит трафика, аналогичного генерируемому при просмотре статей на таких ресурсах, как New York Times, Slashdot и Wired.

Поэтому летом 2001 года участник проекта Wikipedia Magnus Manske (который стал впоследствии студентом университета) начал работу над отдельной системой управления содержимым вебсайта для проекта Wikipedia в свое свободное время. Он решил улучшить производительность программного обеспечения проекта Wikipedia путем использования работающего с базой данных приложения, а также добавить в него специфические для проекта Wikipedia возможности, не предоставляемые "стандартной" системой wiki. Разработанное с использованием языка PHP и базы данных MySQL новое приложение было названо просто "сценарий PHP", "PHP wiki", "программное обеспечение Wikipedia" или "фаза II".

Этот сценарий на языке PHP стал доступен в августе 2001 года, был размещен на сайте SourceForge в сентябре и тестировался до конца 2001 года. Так как проект Wikipedia испытывал сложности из-за повторяющихся проблем с производительностью в условиях растущего трафика, англоязычный раздел энциклопедии Wikipedia в конечном итоге перешел от использования приложения UseModWiki к использованию данного сценария на языке PHP в январе 2002 года. Разделы для других языков, также созданные в 2001 году, также медленно мигрировали, причем некоторые из них использовали приложение UseModWiki до 2004 года.

Так как программное обеспечение, разработанное с использованием языка PHP, работает с базой данных MySQL, сценарий на языке PHP был первым вариантом программного обеспечения, которое впоследствии стало известно под названием MediaWiki. В рамках данного сценария были впервые реализованы такие критические возможности, используемые и сегодня, как пространства имен для организации содержимого (включая страницы обсуждений), оболочки и специальные страницы (включая страницы отчетов об обслуживании, страницы со списком внесенных изменений и списком пользователей).

Фаза III: MediaWiki

Несмотря на улучшения, внесенные благодаря использованию нового сценария на языке PHP с базой данных, комбинация возрастающего объема трафика, ресурсоемких функций и ограниченных аппаратных ресурсов приводила к проблемам с производительностью вебсайта проекта Wikipedia. В 2002 году Lee Daniel Crocker снова переписал код, назвав новое программное обеспечение "Фаза III" (<http://article.gmane.org/gmane.science.linguistics.wikipedia.technical/2794>). Так как

работа сайта периодически нарушалась из-за сложностей, Lee решил, что "времени на основательное проектирование и разработку решения просто нет", поэтому он "просто реорганизовал существующую архитектуру с целью улучшения производительности и переработал весь код". В код были добавлены функции профилирования для отслеживания медленно выполняющихся функций.

Программное обеспечение "Фаза III" использовало тот же основной интерфейс и было спроектировано таким образом, чтобы выглядеть и взаимодействовать с пользователем в большей степени аналогично программному обеспечению "Фаза II" настолько, насколько это возможно. Также было добавлено несколько таких новых функций, как новая система загрузки файлов, двухсторонние списки различий в изменениях содержимого страницы и ссылки между разделами.

Другие функции были добавлены в течение 2002 года и включали новые страницы обслуживания, а также возможность редактирования с помощью двойного клика. При этом проблемы с производительностью снова начали проявляться. Например, в ноябре 2002 года администраторам пришлось временно отключить функцию записи статистики количества просмотров и сайта, которая приводила к выполнению двух операций записи информации в базу данных при каждом просмотре страницы. Они также временами переводили сайт в режим "только для чтения" для поддержания возможности чтения статей и отключения ресурсоемких страниц обслуживания в течение периодов высокой загрузки сайта из-за проблем с блокировкой таблиц.

В начале 2003 года разработчики обсуждали, нужно ли провести повторное проектирование и изменение архитектуры программного обеспечения перед тем, как станет невозможно бороться с нагрузками, либо продолжить дополнять и улучшать существующую кодовую базу. Они выбрали второй вариант в большей степени из-за того, что большинство разработчиков относились достаточно положительно к существующей кодовой базе и было уверено в том, что будущие улучшения будут достаточны для поддержания темпов роста вебсайта.

В июне 2003 года администраторы добавили второй сервер, являющийся первым сервером базы данных, отделенным от веб-сервера. (Новая машина также являлась веб-сервером для неанглоязычных сайтов проекта Wikipedia.) Балансировка нагрузки между двумя серверами должна была быть настроена позднее в течение текущего года. Администраторы также активировали новую систему кэширования страниц, которая использовала файловую систему для хранения сформированных, готовых к отправке страниц для анонимных пользователей.

Также в июне 2003 года Jimmy Wales создал некоммерческую организацию Wikimedia Foundation для поддержки проекта Wikipedia и управления его инфраструктурой, а также выполнения повседневных операций. Программное обеспечение проекта Wikipedia в июле приобрело официальное название "MediaWiki", полученное в результате игры со словами из названия организации Wikimedia Foundation. В то время считалось, что сложные названия могут смутить пользователей и разработчиков.

В июле в программное обеспечение были добавлены такие новые функции, как автоматически генерируемые оглавления и возможность редактирования разделов в рамках страниц, причем обе эти функции используются и сегодня. Первый выпуск приложения с названием "MediaWiki" состоялся в августе 2003 года и завершил становление приложения, структура которого останется относительно стабильной до сегодняшнего дня.

12.2. Кодовая база и практика разработки приложения MediaWiki

PHP

Фреймворк PHP был выбран для разработки программного обеспечения проекта Wikipedia в ходе работы над приложением "Фаза II" в 2001 году; с того времени приложение MediaWiki органично развивалось и развивается до сих пор. Большинство разработчиков проекта MediaWiki являются добровольцами, работающими над приложением в свое свободное время, а в начале развития проекта их было очень мало. Некоторые архитектурные решения и исключения при взгляде из сегодняшнего дня могут показаться некорректными, но сложно критиковать создателей приложения за отсутствие реализации некоторой абстракции, которая является критичной сегодня, в момент, когда кодовая база была достаточно мала, а затраченное на ее разработку время было ограничено.

Например, MediaWiki использует имена классов без префиксов, которые могут привести к конфликтам в момент, когда разработчики PHP или PECL (Библиотека расширений сообщества PHP - PHP Extension Community Library) добавляют новые классы: класс приложения MediaWiki Namespace должен быть переименован в MWNamespace для совместимости с PHP 5.3. Постоянное использование префикса для всех классов (т.е., "MW") должно упростить включение кода приложения MediaWiki в состав другого приложения или библиотеки.

Использование языка PHP было, возможно, не самым лучшим решением в плане производительности, так как он не использует оптимизаций, реализованных в некоторых других динамических языках программирования. Использование языка Java позволило бы получить гораздо лучшую производительность и упростить процесс масштабирования для обслуживания оборудования. С другой стороны, язык PHP очень популярен и его использование упрощает привлечение новых разработчиков.

Даже если приложение MediaWiki все еще содержит "некачественный" устаревший код, значительные улучшения проводились в течение многих лет и новые элементы архитектуры вводились в состав приложения MediaWiki в течение всей истории его развития. Эти улучшения включают в себя классы Parser, SpecialPage и Database, класс Image и иерархию классов FileRepo, иерархии классов ResourceLoader и Action. Приложение MediaWiki начало свое существование без всех этих классов, но все они реализуют функции, которые были доступны с самого начала. Многие разработчики заинтересованы в первую очередь в разработке новых функций и обычно оставляют в стороне вопросы, касающиеся архитектуры, чтобы обратить внимание на них только после окончания разработки, когда отсутствие подходящих архитектурных решений является очевидным.

Безопасность

Так как приложение MediaWiki является платформой для таких известных сайтов, как Wikipedia, основные разработчики и рецензенты кода следуют жестким правилам безопасности. (Ознакомьтесь с [подробным руководством](#).) Для упрощения написания безопасного кода приложение MediaWiki предоставляет разработчикам функции-обертки для доступа к формируемым документам HTML и осуществления запросов к базам данных с удалением управляющих символов. Для нормализации введенных пользователем данных разработчик использует класс WebRequest, который анализирует переданные в составе URL или с помощью формы с POST-запросом данные. Он удаляет "магические кавычки" и слеши, убирает некорректные введенные символы и нормализует последовательности символов Unicode. Атаки на основе межсайтового создания запросов (cross-site request forgery - CSRF) отсекаются путем использования токенов, а атаки на основе межсайтового скрипtingа (cross-site scripting - XSS) - путем проверки вводимых символов и удаления управляющих символов из выводимых последовательностей данных, обычно с помощью функции htmlspecialchars() из состава PHP. Приложение MediaWiki также предоставляет (и использует) систему проверки структуры XHTML, реализованную в рамках класса Sanitizer, а также функции работы с базой данных для предотвращения атак на основе SQL-инъекций.

Конфигурация

Приложение MediaWiki использует тысячи настроек, хранящихся в глобальных переменных PHP. Их значения по умолчанию хранятся в файле `DefaultSettings.php`, а системный администратор может изменить их значения, отредактировав файл `LocalSettings.php`.

MediaWiki в значительной степени зависит от глобальных переменных, включая переменные для хранения настроек и работе с контекстом. Глобальные переменные могут влиять на безопасность приложения в зависимости от использования функции `register_globals` из состава PHP (которая не требуется MediaWiki начиная с версии 1.2). Эта система также ограничивает возможности абстракций конфигурации и затрудняет оптимизацию процесса запуска. Более того, пространство имен конфигурации делится с переменными, используемыми для регистрации пользователей и объектов для управления контекстом, что ведет к потенциальным конфликтам. С точки зрения пользователя, глобальные переменные для конфигурации сделали приложение MediaWiki на первый взгляд сложным для настройки и сопровождения. Процесс разработки MediaWiki стал историей медленного перемещения элементов контекста из глобальных переменных в состав объектов. Хранение элементов управления контекстом в переменных объектов позволяет осуществлять более гибкое повторное использование этих объектов.

12.3. База данных и хранилище текста

Приложение MediaWiki использует реляционную базу данных с момента создания программного обеспечения с названием "Фаза II". Стандартной (и поддерживаемой лучшим образом) системой управления базами данных (database management system - DBMS) для MediaWiki является система MySQL, используемая всеми сайтами организации Wikimedia, но другие системы управления базами данных (такие, как PostgreSQL, Oracle и SQLite) поддерживаются силами сообщества. Системный администратор может выбрать систему управления базами данных в процессе установки приложения MediaWiki, при этом MediaWiki предоставляет и абстракцию для базы данных, и абстракцию уровня запросов, которая упрощает доступ к базе данных для разработчиков.

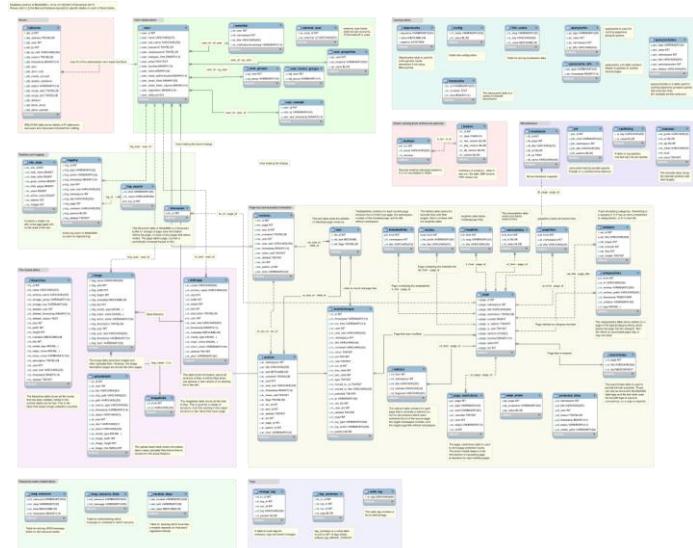


Рисунок 12.1: Схема базы данных

В данный момент база данных содержит множество таблиц. Многие таблицы относятся к функциям хранения содержимого wiki (т.е., таблицы `page`, `revision`, `category` и `recentchanges`). Другие таблицы содержат данные пользователей (таблицы `user`, `user_groups`), мультимедийных файлов (`image`, `filearchive`), кэширования (`objectcache`, `110n_cache`, `querycache`) и инструментов для внутренних операций (таблица `job` для хранения очередей задач), а также другие данные, как показано на [Рисунке 12.2](#). (Доступна [документация с полным описанием структуры базы данных MediaWiki](#).) Индексы и итоговые таблицы интенсивно используются MediaWiki, так как SQL-запросы, затрагивающие большие количества строк, могут оказаться чрезвычайно ресурсоемкими, особенно в случае сайтов организации Wikimedia. Запросы без индексов обычно отклоняются.

В течение многих лет было проведено большое количество изменений схемы базы данных, при этом наиболее важным было разделение хранилища текстовых данных и данных изменений в MediaWiki версии 1.5.

```

cur:          page:
  cur_id      page_id
  cur_namespace page_namespace
  cur_title    page_title
  cur_text     page_restrictions
  cur_comment   page_counter
  cur_user     page_is_redirect
  cur_user_text page_is_new
  cur_timestamp page_random
  cur_restrictions page_touched
  cur_counter   page_latest
  cur_is_redirect
  cur_minor_edit
  cur_is_new
  cur_random
  cur_touched
  inverse_timestamp

old:
  old_id      revision:
  old_namespace rev_id
  old_title    rev_page
  old_text     rev_comment
  old_comment   rev_user
  old_user     rev_user_text
  old_timestamp rev_timestamp
  old_minor_edit
  old_flags    inverse_timestamp
  old_text
  old_id
  old_text
  old_flags
  inverse_timestamp

```

Рисунок 12.2. Основные таблицы данных в MediaWiki версий 1.4 и 1.5

В модели базы данных версии 1.4 содержимое страниц хранилось в двух важных таблицах: `cur` (содержащей текст и метаданные последней ревизии страницы) и `old` (содержащей данные предыдущих ревизий); удаленные страницы хранились в таблице `archive`. Когда выполнялось редактирование, действующая ревизия копировалась в таблицу `old`, а новая ревизия сохранялась в таблицу `cur`. При переименовании страницы заголовок страницы обновлялся в метаданных всех устаревших ревизий из таблицы `old`, что было достаточно длительной операцией. При удалении страницы все ее элементы из таблиц `cur` и `old` должны были копироваться в таблицу `archive` перед самим удалением; эта операция подразумевала перемещение текстовых данных всех ревизий, которые могли иметь большой объем и, таким образом, выполнение операции могло занимать большой промежуток времени.

В модели базы данных версии 1.5 метаданные и текст ревизий были разделены: таблицы `cur` и `old` были заменены на таблицы `page` (для хранения метаданных страниц), `revision` (для хранения метаданных всех ревизий, устаревших или действующих) и `text` (для хранения текстовых данных всех ревизий, устаревших и действующих или удаленных). Теперь в случае редактирования метаданные ревизии не должны копироваться между таблицами: вполне достаточно вставки нового элемента и обновления указателя `page_latest`. Также, метаданные ревизии больше не включают в себя заголовка страницы, а содержат только ее идентификатор: это обстоятельство исключает необходимость переименования всех ревизий при переименовании страницы.

Таблица `revision` содержит метаданные для каждой ревизии, но не текст ревизий; наоборот, в составе метаданных содержится идентификатор текста, указывающий на элемент таблицы `text`, содержащий соответствующие текстовые данные. При удалении страницы текст всех ревизий страницы остается на том же месте и его перемещения в другую таблицу попросту не требуется. Таблица `text` содержит соответствующие идентификаторам текстовые данные; поле `flags` указывает на то, сжаты ли текстовые данные с помощью `gzip` (для экономии дискового пространства) или на то, являются ли данные простым указателем на внешнее хранилище текстовых данных. Сайты организации Wikimedia используют кластер для формирования внешнего хранилища данных на основе системы MySQL и хранения в нем данных множества ревизий. Первая ревизия текстовых данных хранится в полном объеме, а следующие ревизии той же страницы хранятся в форме спи-

ска различий новой и предыдущей ревизии; впоследствии данные сжимаются с помощью gzip. Так как ревизии группируются в соответствии со страницами, они схожи, поэтому списки различий относительно малы и сжатие с помощью gzip отлично работает. Степень сжатия, которая достигается при работе с сайтами организации Wikimedia, составляет около 98%.

На стороне аппаратного обеспечения приложение MediaWiki применяет встроенную систему балансировки нагрузки, добавленную в состав приложения в 2004 году во время выпуска MediaWiki версии 1.2 (когда проект Wikipedia получил в свое распоряжение второй сервер - значительное приобретение в то время). Балансировщик нагрузки (код PHP в составе приложения MediaWiki, который решает, с каким сервером соединиться) на данный момент является критической частью инфраструктуры организации Wikimedia, которая оказывает влияние на некоторые решения, реализованные в форме алгоритмов из кода. Системный администратор может установить в файле конфигурации приложения MediaWiki один ведущий сервер и любое количество ведомых серверов баз данных; значение приоритета может быть установлено для каждого из серверов. Балансировщик нагрузки будет переадресовывать все операции записи ведущему серверу и балансировать операции чтения в зависимости от приоритетов серверов. Он также следит за задержкой при копировании данных каждым из ведомых серверов. Если задержка при записи ведомым сервером превышает 30 секунд, он не будет получать каких-либо очередей чтения, что позволит завершить выполнение операции; если все ведомые серверы испытывают задержки более 30 секунд, приложение MediaWiki автоматически переведет себя в режим работы с выполнением исключительно операций чтения данных.

Система "хронологической защиты" приложения MediaWiki позволяет быть уверенными в том, что задержка при копировании данных никогда не приведет к тому, что пользователю будет показана страница с информацией, при ознакомлении с которой можно будет сделать вывод о том, что недавно выполненная операция еще не завершилась: например, если пользователь переименовывает страницу, другой пользователь может все еще видеть ее старое название, но тот, кто ее переименовал всегда будет видеть новое имя, так как он является тем, кто ее переименовал. Это поведение реализуется путем сохранения данных о ведущем сервере в рамках пользовательской сессии в том случае, если выполненный запрос привел к отправке запроса записи к базе данных. В следующий раз, когда пользователь осуществит запрос чтения, балансировщик нагрузки получит данные о расположении сервера из данных сессии и попытается выбрать ведомый сервер, на который были скопированы данные, после чего выполнить запрос. Если сервер не доступен, система будет ожидать ввода сервера в эксплуатацию. Другим пользователям может казаться, что действие еще не выполнено, но хронология остается последовательной для каждого из пользователей.

12.4. Запросы, кэширование и доставка данных

Процесс выполнения веб-запроса

Файл index.php является основной точкой входа приложения MediaWiki и принимает большинство запросов, обрабатываемых серверами приложения (т.е., запросов, не обрабатываемых инфраструктурой кэширования, о которой написано ниже). Исполняемый код из файла index.php производит все проверки безопасности данных, загружает стандартные параметры настройки из файла includes/DefaultSettings.php, устанавливает параметры настройки с помощью файла исходного кода includes/Setup.php, после чего применяет параметры настройки вебсайта, заданные в файле LocalSettings.php. После этого создается объект MediaWiki (\$mediawiki), а также объект Title (\$wgtitle), в зависимости от параметров, задающих заголовок и действие в рамках запроса.

Файл index.php может принимать множество описывающих действие параметров в рамках запроса с использованием URL; стандартным параметром является параметр view, с помощью которого производится обычный вывод содержимого статьи. Например, запрос <https://en.wikipedia.org/w/index.php?title=Apple&action=view> выводит содержимое статьи с названием "Apple" из англоязычной энциклопедии Wikipedia. (Запросы просмотра статей обычно упрощают

щаются с помощью механизма перезаписи URL, а в данном случае URL примет следующий вид: <https://en.wikipedia.org/wiki/Apple>.) Другими часто используемыми параметрами являются `edit` (применяемый для открытия статьи с целью редактирования), `submit` (для предварительного просмотра или сохранения статьи), `history` (для показа истории редактирования статьи) и `watch` (для добавления статьи в пользовательский список наблюдения). Административные действия выполняются с помощью параметров `delete` (для удаления статьи) и `protect` (для запрета редактирования статьи).

Функция `MediaWiki::performRequest()` вызывается впоследствии для выполнения наибольшего количества работы, связанной с обработкой запроса в форме URL. Она проверяет наличие некорректных заголовков, ограничений чтения, локальных перенаправлений и циклов перенаправлений, а также устанавливает, направлен ли запрос на показ обычной или специальной страницы.

Запросы обычных страниц осуществляются с помощью вызовов функции

`MediaWiki::initializeArticle()` для создания объекта статьи `Article` для страницы (`$wgArticle`), после чего вызывается функция `MediaWiki::preformAction()`, которая выполняет "стандартные" действия. Как только выполнение действия завершается, функция `MediaWiki::finalCleanup()` завершает выполнение запроса, принудительно выполняя транзакции на уровне базы данных, выводя HTML-документ и запуская отложенные обновления данных из очереди задач. Функция `MediaWiki::restInPeace()` осуществляет выполнение отложенных обновлений данных и корректно завершает выполнение задачи.

Если запрашиваемая страница является специальной страницей (т.е., не обычной страницей `wiki` с содержимым, а такой специальной страницей с подробностями о функционировании программного обеспечения, как страница статистики `Statistics`), вместо функции `initializeArticle()` вызывается функция `SpecialPageFactory::executePath()`; впоследствии выполняется соответствующий сценарий PHP. Специальные страницы позволяют выполнять все типы нестандартных задач, при этом каждая страница имеет специфическую цель, обычно не зависящую от любой из статей, а также содержимого этой статьи. Специальные страницы, среди прочего, позволяют знакомиться с различными типами отчетов (списками недавних обновлений, журналами событий, страницами без категорий) а также работать с инструментами администрирования `wiki` (блокировать пользователей, изменять права пользователей). Их принцип работы зависит от их функций.

Многие функции содержат код профилирования, который делает возможным отслеживание хода исполнения функций для отладки в случае включения профилирования. Профилирование осуществляется с помощью вызовов функций `wfProfileIn` и `wfProfileOut`, которые включают и отключают функцию профилирования соответственно; обе функции принимают в качестве параметра имя функции. На сайтах организации Wikimedia с целью сохранения производительности профилирование производится только с использованием некоторого процента всех запросов. Приложение MediaWiki отправляет UDP-пакеты центральному серверу, который накапливает их и формирует данные профилирования на их основе.

Кэширование

Приложение MediaWiki оптимизировано с целью повышения производительности, так как оно играет ключевую роль в функционировании вебсайтов организации Wikimedia, а также из-за того, что оно является частью большой функционирующей экосистемы, которая повлияла на его архитектуру. Инфраструктура кэширования данных организации Wikimedia (разделенная на уровни) наложила ограничения на возможности приложения MediaWiki; разработчики пытались устранить проблемы, не стараясь подстроиться под принцип работы в значительной степени оптимизированной инфраструктуры кэширования организации Wikimedia, сформированной вокруг приложения MediaWiki, а делая приложение MediaWiki более гибким таким образом, чтобы оно могло работать в рамках этой инфраструктуры без ущерба требуемым возможностям производительности и кэширования. Например, по умолчанию приложение MediaWiki выводит IP-адрес пользователя в

правом верхнем углу страницы (для языков с написанием слева направо) в качестве напоминания о том, как как пользователи идентифицируются программным обеспечением в период работы с системой. Переменная конфигурации `$wgShowIPHeader` позволяет системному администратору отключить эту возможность, делая тем самым содержимое страницы независимым от пользователя: все анонимные посетители смогут получать одну и ту же версию каждой страницы.

Первый уровень кэширования (используемый на сайтах организации Wikimedia) состоит из прокси-серверов обратного кэширования (Squid), которые перехватывают и выполняют большинство запросов до момента их обработки с помощью серверов приложения MediaWiki. Серверы Squid содержат статические версии полностью сформированных страниц, пригодных для чтения пользователями, не совершившими вход в систему. Приложение MediaWiki изначально поддерживает механизмы взаимодействия с серверами Squid и Varnish, а также интегрирует их в систему уровня кэширования для выполнения таких действий, как, например, отправка уведомления им о необходимости удаления страницы из кэша после ее изменения. Для пользователей, совершивших вход в систему, а также других запросов, которые не могут обрабатываться с помощью серверов Squid, кэширующие серверы пересыпают запросы напрямую веб-серверу (Apache).

Второй уровень кэширования реализуется в ходе создания и формирования страницы приложением MediaWiki из множества объектов, многие из которых могут кэшироваться для сокращения количества последующих вызовов. Эти объекты включают в себя интерфейс страницы (боковую панель, меню, текст пользовательского интерфейса) и ее содержимое, сформированное в ходе разбора текстовых данных с разметкой wiki (wikitext). Система кэширования объектов в оперативной памяти была доступна в приложении MediaWiki начиная с версии 1.1 (выпущенной в 2003 году) и очень важна для предотвращения повторных разборов больших и сложных страниц.

Данные сессии при входе в систему также могут сохраняться с помощью системы Memcached, что позволяет организовать прозрачную работу механизма сессий в окружении, состоящем из множества серверов с системой балансировки нагрузки (инфраструктура организации Wikimedia формировалась с расчетом на использование системы балансировки нагрузки LVS совместно с PyBal).

Начиная с версии 1.16 приложение MediaWiki использует отдельный кэш объектов для хранения локализованных строк пользовательского интерфейса; этот кэш был добавлен после выявления того, что большое количество объектов, сохраняемых с помощью системы Memcached, представляет собой локализованные с учетом используемого языка сообщения пользовательского интерфейса. Система кэширования реализует возможность быстрого получения отдельных сообщений из баз данных констант (constant databases - CDB), т.е., файлов, содержащих пары ключ-значение. Базы данных констант позволяют снизить затраты памяти и время запуска системы в стандартных условиях; они также используются для функционирования кэшей уровня wiki.

Последний уровень кэширования представлен системой кэширования байткода PHP, обычно активируемой для ускорения приложений на языке PHP. Компиляция может быть длительной; для преодоления необходимости компиляции сценариев PHP в байткод каждый раз при их запуске может быть использован ускоритель PHP, позволяющий хранить скомпилированный байткод и выполнять его непосредственно без компиляции. Приложение MediaWiki будет "просто работать" совместно с многими ускорителями, такими, как APC, PHP accelerator и eAccelerator.

Ввиду большой нагрузки на инфраструктуру организации Wikimedia, приложение MediaWiki было оптимизировано для работы с завершенной многослойной распределенной инфраструктурой кэширования данных. Несмотря на это, приложение также предоставляет возможность использования дополнительной упрощенной системы кэширования, использующей файловую систему для хранения полностью сформированных выводимых страниц аналогично кэширующему серверу Squid. Также абстрактный уровень кэширования объектов приложения MediaWiki позволяет хранить объекты в нескольких местах, включая файловую систему, базу данных или кэш байткода.

Модуль ResourceLoader

Как и в случае многих других веб-приложений, интерфейс приложения MediaWiki становился более быстрым и отзывчивым в течение многих лет, причем в большей степени это стало возможным благодаря использованию языка JavaScript. Улучшение пользовательских качеств приложения было начато в 2008 году вместе с разработкой системы для расширенной работы с мультимедийными файлами (т.е., системы для реализации таких функций, как редактирование видеофайлов с помощью веб-приложения), создававшейся для улучшения производительности удаленного пользовательского интерфейса.

Для оптимизации процесса доставки данных сценариев на языке JavaScript и стилей CSS был разработан модуль ResourceLoader, оптимизирующий процесс доставки данных JS и CSS. Начатая в 2009 году разработка была завершена в 2011 году и стала основной функцией приложения MediaWiki начиная с версии 1.17. Модуль ResourceLoader осуществляет доставку данных JS и CSS по требованию, таким образом экономя время, затрачиваемое на загрузку и разбор данных в случаях отсутствия необходимости в этих операциях, например, при использовании устаревших браузеров. Также данный модуль позволяет уменьшить объем кода, сгруппировать ресурсы для уменьшения количества запросов и, кроме того, вставлять изображения с помощью строк URI для данных. (Для получения более подробной информации о модуле ResourceLoader следует обратиться к [официальной документации](#) и выступлению Trevor Parscal и Roan Kattouw с названием "*Low Hanging Fruit vs. Micro-optimization: Creative Techniques for Loading Web Pages Faster*" на конференции OSCON 2011.)

12.5. Языки

Контекст и обоснование

Ключевым фактором успешного процесса создания и распространения свободной информации с участием всех желающих является предоставление этой информации на стольких языках, на скольких это возможно. Сайт Wikipedia доступен на более чем 280 языках и англоязычные статьи энциклопедии составляют менее чем 20% от всех статей. Так как сайт Wikipedia и родственные ему сайты существуют на таком огромном количестве языков, важно не только предоставлять читателям содержимое статей на их родном языке, но также предоставлять в их распоряжение локализованный интерфейс и эффективные инструменты ввода и преобразования текста для того, чтобы участники проекта могли добавлять информацию.

По этой причине системы локализации и интернационализации (110n и i18n) являются ключевыми компонентами приложения MediaWiki. Система интернационализации производит глубокие изменения данных и затрагивает большое количество программных компонентов; она также является одной из наиболее гибких и функциональных систем. (Существует [исчерпывающее руководство](#) по интернационализации и локализации приложения MediaWiki.) Удобство переводчиков обычно оказывается предпочтительнее удобства разработчиков, хотя и считается, что обе группы должны находятся в одинаковых условиях.

Приложение MediaWiki на данный момент локализовано с использованием более чем 350 языков, включая не латинские языки и языки с написанием справа налево (RTL), причем локализации имеют различный статус завершения. Интерфейс и содержимое могут использовать разные языки, а также быть смешаны.

Язык содержимого статей

Изначально приложение MediaWiki использовало кодировки в зависимости от используемых языков, что приводило к множеству проблем; например, сценарии, использующие другие языки, не могли использоваться в заголовках страниц. Вместо различных кодировок была применена коди-

ровка UTF-8. Поддержка отличных от UTF-8 наборов символов была прекращена в 2005 году вместе с кардинальными изменениями схемы базы данных в версии MediaWiki 1.5; в данный момент текст статей должен использовать кодировку UTF-8.

Символы, не доступные на клавиатуре редактирующего статью участника, могут быть заданы и вставлены с помощью инструментов редактирования приложения MediaWiki (Edittools), элемента интерфейса, расположенного ниже окна редактирования; версия этих инструментов, работающая с использованием языка JavaScript, автоматически вставляет выбранный символ в окно редактирования. Расширение WikiEditor для приложения MediaWiki, разработанное в рамках кампании по улучшению пользовательских качеств приложения, объединяет список специальных символов с панелью инструментов редактирования. Другое расширение с названием Narayam предоставляет возможность использования дополнительных методов ввода и возможностей назначения клавиш для символов, не входящих в таблицу ASCII.

Язык интерфейса

Сообщения интерфейса хранились в массивах PHP в форме пар ключ-значение начиная с момента создания программного обеспечения под названием "Фаза III". Каждое сообщение идентифицируется с помощью уникального ключа, который ставится в соответствие различным значениям для различных языков. Ключи задаются разработчиками, вынужденными использовать префиксы для расширений; например, ключи сообщений для расширения UploadWizard начинаются с префикса `mwe-uowiz-`, где `mwe` расшифровывается как расширение MediaWiki (*MediaWiki extension*).

Сообщения приложения MediaWiki могут включать в свой состав заданные программным обеспечением параметры, которые обычно влияют на грамматику сообщения. В общем, для теоретической поддержки любого существующего языка система локализации приложения MediaWiki со временем была усовершенствована и усложнена с целью адаптации к специфичным для языков особенностям и исключениям, которые обычно кажутся странными англоговорящей аудитории.

Например, определения являются неизменяемыми словами в английском языке, но в таких языках, как французский требуется согласование определений с существительными. Если пользователь указал свой пол на странице настроек профиля, для корректного обращения к пользователю в сообщениях интерфейса может использоваться модификатор `{{GENDER:}}` . Множество других модификаторов включает в свой состав модификатор `{{PLURAL:}}` для "простых" языковых групп и языков, таких, как арабский, использующих двойные, тройные или краткие записи чисел, а также модификатор `{{GRAMMAR:}}` , позволяющий выполнять функции грамматических преобразований для таких языков, как финский, грамматические особенности которого предполагают наличие различий или изменений форм слов.

Локализация сообщений

Локализованные сообщения интерфейса приложения MediaWiki находятся в файлах `MessagesXx.php`, где `Xx` является кодом ISO-639 для языка (т.е., для французского языка файл будет иметь название `MessagesFr.php`); используемые по умолчанию сообщения написаны на английском языке и находятся в файле `MessagesEn.php`. Расширения приложения MediaWiki используют аналогичную систему или хранят все локализованные сообщения в файле с именем `<Название-расширения>.i18n.php`. Вместе с переродами, файлы сообщений также включают такую специфическую для языков информацию, как форматы даты.

Передача переводов проекту обычно осуществляется путем отправки патчей для файлов исходного кода PHP с именами `MessagesXx.php`. В декабре 2003 года в рамках выпуска версии 1.1 приложения MediaWiki была представлена система "сообщений из базы данных", представляющая собой подмножество страниц `wiki` из пространства имен MediaWiki, хранящих сообщения интерфейса. Содержимое страницы `wiki` с названием `MediaWiki:<Ключ-сообщения>` является текстом сооб-

щения, более приоритетным, чем значение из файла сообщений PHP. Локализованные версии сообщения хранятся в форме страниц с названиями MediaWiki:<Ключ-сообщения>/<код-языка>; например, MediaWiki:Rollbacklink/de.

Эта возможность позволила опытным пользователям перевести (и изменить) сообщения интерфейса в рамках своей системы wiki, в ходе процесса не модифицируя поставляемые в составе приложения MediaWiki файлы интернационализации. В 2006 году Niklas Laxstrom создал специальный, значительно доработанный сайт на основе MediaWiki (на сегодняшний день располагающийся по адресу <http://translatewiki.net>), на котором переводчики могли без лишних сложностей локализовать сообщения интерфейса для всех языков, просто редактируя страницу wiki. После этого обновлялись файлы messagesXX.php в репозитории исходного кода приложения MediaWiki, из которого они могли автоматически извлекаться любой системой wiki, а также обновляться с помощью расширения LocalisationUpdate. На сайтах организации MediaWiki сообщения из базы данных используются в данный момент только для изменения вида страниц, но не для локализации. Расширения приложения MediaWiki и некоторые такие сопутствующие приложения, как боты также локализуются с помощью сайта translatewiki.net.

Для облегчения понимания переводчиками контекста и значения сообщения интерфейса при разработке приложения MediaWiki хорошей практикой является предоставление документации для каждого сообщения. Эта документация хранится в специальном файле сообщений с кодом языка qqq, который не соответствует какому-либо из реально существующих языков. Документация для каждого сообщения выводится в интерфейсе перевода на сайте translatewiki.net. Другим полезным инструментом является код языка qqx; при его использовании в качестве аргумента параметра &uselang после запроса страницы wiki (т.е., при использовании запроса, аналогичного <https://en.wikipedia.org/wiki/Special:RecentChanges?uselang=qqx>) приложение MediaWiki выведет имена ключей сообщений вместо их значений при формировании пользовательского интерфейса; это очень полезно для идентификации того, какое сообщение следует перевести или изменить.

Зарегистрированные пользователи могут выбрать свой язык интерфейса на странице настроек в случае необходимости изменения стандартного языка интерфейса вебсайта. Приложение MediaWiki также поддерживает языки, выбираемые в случае неполадок: если сообщение не доступно для выбранного языка, будет выведено сообщение для наиболее схожего языка, который не обязательно должен являться английским. Например, в качестве замены бретонского языка используется французский язык.

12.6. Пользователи

Пользователи представлены в коде с помощью экземпляров класса User, который инкапсулирует все специфические для пользователя настройки (идентификатор, имя, права доступа, пароль, адрес электронной почты, и.т.д.). Клиентские классы используют специальные функции для доступа к этим полям; они выполняют всю работу по определению того, осуществил ли пользователь вход в систему и может ли быть установлено значение запрашиваемого параметра с помощью кук или необходим запрос к базе данных. Большая часть параметров, требуемых для формирования стандартных страниц, сохраняется в куках для сокращения количества запросов к базе данных.

Приложение MediaWiki предоставляет очень неоднородную систему прав доступа с устанавливаемыми правами пользователя, в общем, для выполнения любых возможных действий. Например, для выполнения "отката" (т.е., для "быстрой отмены всех операций редактирования последним пользователем определенной страницы") пользователю требуется разрешение с названием "rollback", выданное по умолчанию для группы пользователей "sysop" приложения MediaWiki. Но это разрешение может быть выдано и другим пользовательским группам или исключительно для данного разрешения может быть выделена отдельная группа (этот подход используется в англоязычной энциклопедии Wikipedia в рамках группы Rollbackers). Изменение прав пользователя

осуществляется путем редактирования массива \$wgGroupPermissions в файле LocalSettings.php; например, объявление \$wgGroupPermissions['user'][]['movefile'] = true; позволяет всем зарегистрированным пользователям осуществлять переименование файлов. Пользователь может быть членом нескольких групп и наследовать наиболее важные права каждой из них.

Однако, система прав пользователей приложения MediaWiki проектировалась с учетом особенностей сайта Wikipedia: сайта с доступным для всех содержимым и с запретом только некоторых действий для некоторых пользователей. В приложении MediaWiki отсутствует унифицированная концепция всеобъемлющих прав доступа; оно не предоставляет таких традиционных возможностей систем управления содержимым вебсайта, как запрет доступа для чтения или записи при указании темы или типа содержимого. Несколько расширений приложения MediaWiki предоставляют такие возможности с некоторыми ограничениями.

12.7. Содержимое статей

Структура содержимого статей

Концепция пространств имен была использована в течение периода эксплуатации приложения UseModWiki сайтом Wikipedia, причем страницы обсуждений имели заголовок "<Название статьи>/Talk". Формально пространства имен были реализованы Magnus Manske в первом "сценарии PHP". Они были несколько раз повторно реализованы в течение длительного промежутка времени, но сохранили свое предназначение: разделение различных типов содержимого страниц. Они состоят из префикса, отделенного от имени страницы двоеточием (т.е., Talk: или File: и Template:); пространство имен позволяет не использовать префикс для доступа к основному содержимому страницы. Пользователи сайта Wikipedia быстро начали использовать эту возможность и создали сообщество с большим количеством возможностей для участия. Пространства имен доказали необходимость своего существования в качестве функции приложения MediaWiki, так как с помощью них создавались необходимые условия для организации пространства дискуссий сообщества wiki, процессов в рамках сообщества, порталов, профилей пользователей, и.т.д.

Стандартные настройки пространства имен основного содержимого страницы приложения MediaWiki предполагают плоскую организацию (без подстраниц), так как таким образом функционирует сайт Wikipedia, но включить поддержку подстраниц достаточно просто. Они включены в других пространствах имен (т.е., пространстве имен User:, где пользователи могут, например, работать с черновиками статей) и обозначаются специальным образом.

Пространства имен разделяют содержимое на основе типов; в рамках одного пространства имен страницы могут быть организованы на основе тем с использованием категорий, эксплуатируя псевдо-иерархическую схему организации, представленную в версии 1.3 приложения MediaWiki.

Обработка содержимого статей: язык разметки MediaWiki и механизм его разбора

Создаваемое пользователями содержимое страниц сохраняется приложением MediaWiki с использованием не формата разметки HTML, а специфического для MediaWiki языка разметки, иногда называемого "wikitext". Этот язык разметки позволяет пользователям изменять форматирование текста (т.е., делать текст жирным или наклонным с использованием кавычек), добавлять ссылки (с помощью квадратных скобок), подключать шаблоны, включать в состав текста зависимое от контекста содержимое (аналогичное дате или подписи), а также выполнять огромное количество других замечательных вещей. ([Подробная документация доступна.](#))

Для вывода страницы ее содержимое должно быть разобрано и дополнено путем сборки всех внешних или динамических элементов, вызываемых данной страницей, после чего должна быть произведена конвертация в корректное представление документа HTML. Система разбора языка

разметки является одной из наиболее важных частей приложения MediaWiki, которую сложно изменить или улучшить. Так как возможность вывода в формате HTML миллионов страниц по всему миру зависит от системы разбора языка разметки, эта система является чрезвычайно стабильной.

Язык разметки не был формально описан с самого начала; он был создан на основе языка разметки приложения UseModWiki, а после этого изменялся и развивался в соответствии с предъявляемыми к нему требованиями. В условиях отсутствия формальной спецификации, язык разметки приложения MediaWiki превратился в сложный и своеобразный язык, полностью совместимый исключительно с системой разбора языка разметки приложения MediaWiki; он не может быть представлен формальной грамматикой. О спецификация применяемой на данный момент системы разбора языка разметки в шутку упоминают, как об "описании всего того, что система разбора языка разметки извлекает из данных в формате wikitext с добавлением описания нескольких сотен вариантов тестирования".

Было предпринято множество попыток разработки альтернативных систем разбора языка разметки, но ни одна из них не увенчалась успехом. В 2004 году экспериментальная система разделения текста была разработана Jens Frank с целью разбора данных в формате wikitext и впоследствии применена на сайте Wikipedia; ее пришлось отключить после трех дней эксплуатации из-за низкой производительности системы резервирования участков памяти для массивов языка PHP. С того времени большая часть задач по разбору языка разметки производилась с использованием огромного количества регулярных выражений и множества вспомогательных функций. Разметка wiki, а также множество специальных случаев, которые система ее разбора должна учитывать, стали относительно более сложными, что еще в большей степени затруднило последующие попытки разработки системы разбора языка разметки.

Важным улучшением являлась выполненная Tim Starling переработка кода препроцессора, осуществленная в версии 1.12 приложения MediaWiki, причем основной мотивацией разработчика было желание увеличить производительность разбора данных страниц со сложными шаблонами. Препроцессор конвертирует данные в формате wikitext в представляющее части документа (включающее вызовы шаблонов, функции системы разбора языка разметки, функции тэгов, заголовки разделов и несколько других структур) дерево XML DOM с пропуском таких "неиспользуемых ветвей", как операторы `#switch` без последующих условий и неиспользуемые стандартные значения для аргументов шаблона при его разборе. После этого система разбора языка разметки обходит структуру DOM и преобразует ее данные в формат HTML.

Недавняя работа над визуальным редактором для приложения MediaWiki привела к необходимости усовершенствования процесса разбора языка разметки (и его ускорения), таким образом была вновь начата работа над системой разбора языка разметки и над промежуточными уровнями представлений в диапазоне между языком разметки MediaWiki и окончательным документом в формате HTML (обратитесь к разделу "Планы на будущее", расположенному ниже).

Специальные слова и шаблоны

Приложение MediaWiki позволяет использовать "специальные слова", с помощью которых изменяется стандартное отображение страницы или в ее состав включаются динамические элементы. Специальные слова делятся на категории: модификаторы поведения `_NOTOC_` (для автоматического скрытия оглавления) или `_NOINDEX_` (для сообщения поисковым машинам о том, что страницы не должна индексироваться); переменные, такие, как `{CURRENTTIME}` или `{SITENAME}`; и функции системы разбора языка разметки, т.е., специальные слова, которые могут принимать параметры, такие, как `{lc:<string>}` (для вывода строки, передаваемой в качестве параметра `<string>`, в нижнем регистре). Такие конструкции, как `{GENDER:}`, `{PLURAL:}` и `{GRAMMAR:}` используются для локализации пользовательского интерфейса и являются функциями системы разбора языка разметки.

Наиболее часто применяемым способом включения содержимого других страниц в состав страницы приложения MediaWiki является использование шаблонов. На самом деле шаблоны были предназначены для включения одной и той же информации в состав различных страниц, т.е., навигационных панелей или баннеров с сообщениями о техническом обслуживании на страницах статей сайта Wikipedia; возможность создания частей страниц и повторного использования их в составе тысяч статей с централизацией управления ими позволила повысить популярность сайтов, подобных Wikipedia.

Однако, шаблоны также использовались (в том числе и чрезмерно) пользователями для достижения совершенно другой цели. В версии 1.3 приложения MediaWiki появилась возможность передачи параметров шаблонам для изменения их содержимого; возможность добавления стандартного значения параметра (введенная в версии 1.6 приложения MediaWiki) позволила реализовать функциональный язык программирования на основе PHP, который в итоге оказался одной из самых ресурсоемких функций приложения.

Впоследствии Tim Starling разработал дополнительные функции системы разбора языка разметки (расширение ParserFunctions) в качестве временной меры для борьбы с абсурдными конструкциями, создаваемыми пользователями сайта Wikipedia с помощью шаблонов. Этот набор функций включал такие логические структуры, как `#if` и `#switch`, а также другие функции, такие, как `#expr` (для вычисления значений математических выражений) и `#time` (для форматирования строки времени).

Спустя достаточно короткий промежуток времени, пользователи сайта Wikipedia начали создавать еще более сложные шаблоны с использованием новых функций, которые сравнительно сильно снизили производительность системы разбора языка разметки при работе со страницами, содержащими большое количество шаблонов. Новый препроцессор, включенный в состав версии 1.12 приложения MediaWiki (в качестве важного архитектурного изменения) был реализован для частичного решения этой проблемы. Не так давно разработчики приложения MediaWiki обсуждали возможность использования существующего языка сценариев, возможно Lua, для улучшения производительности.

Мультимедийные файлы

Пользователи загружают файлы с помощью страницы `Special:Upload`; администраторы могут задать разрешенные типы файлов с помощью списка разрешенных расширений файлов. После загрузки файлы хранятся в директории файловой системы, а миниатюры для предварительного просмотра - в отдельной директории с названием `thumb`.

Так как организация Wikimedia выполняет образовательную миссию, приложение MediaWiki поддерживает типы файлов, которые могут быть нестандартными для других веб-приложений и систем управления содержимым вебсайта, такие, как векторные изображения SVG и многостраничные документы PDF и DjVu. Они представляются с помощью изображений формата PNG и могут быть отображены и включены в состав страницы также, как и более известные типы изображений, такие, как GIF, JPG и PNG.

После загрузки файла ему ставится в соответствие страница `File:`, содержащая информацию, введенную загрузившим файл пользователем; это описание в свободной форме обычно включает информацию о правообладателе (имя автора, лицензия) и элементы, описывающие или классифицирующие содержимое файла (описание, расположение, дата, категории, и.т.д.). В то время, как отдельные установленные системы wiki могут не предъявлять требований к этой информации, в медиа-библиотеках, таких, как Wikimedia Commons крайне важно предоставлять данные для организации коллекций и уверенности в том, что файлы могут легально распространяться. Было аргументировано утверждение о том, что большая часть этих метаданных фактически должна храниться в рамках такой структуры, поддерживающей выполнение запросов, как таблица базы дан-

ных. Это сравнительно упростит не только поиск, но и установление авторства, а также повторное использование материалов сторонними лицами - например, с помощью API.

Большинство сайтов организации Wikimedia позволяет совершать "локальные" загрузки файлов в каждый из разделов wiki, но сообщество пытается хранить распространяемые под свободными лицензиями мультимедийные файлы в свободной медиа-библиотеке организации Wikimedia, Wikimedia Commons. Любой вебсайт организации Wikimedia может отобразить файл, размещенный в Wikimedia Commons также, как в случае его локального размещения. Это правило позволяет преодолеть необходимость загрузки файла в каждый раздел wiki для того, чтобы использовать его там.

В результате приложение MediaWiki изначально поддерживает сторонние репозитории мультимедийных файлов, т.е., возможность доступа к мультимедийным файлам, размещенным в других разделах wiki при помощи API и системы ForeignAPIRepo. Начиная с версии 1.16, любой использующий MediaWiki вебсайт может без лишних сложностей использовать файлы из репозитория Wikimedia Commons с помощью функции InstantCommons. При использовании стороннего репозитория миниатюры для предварительного просмотра хранятся локально с целью экономии пропускной способности сети. Однако, (на данный момент) невозможно загружать файлы в сторонний репозиторий мультимедийных файлов с помощью не относящейся к нему системы wiki.

12.8. Модификации и расширение возможностей MediaWiki

Уровни доступа

Архитектура приложения MediaWiki предусматривает различные способы модификации и расширения возможностей программного обеспечения. Эти операции могут быть выполнены при различных уровнях доступа:

- Системные администраторы могут устанавливать расширения и оболочки, настраивать отдельные вспомогательные программы wiki (т.е., программы для создания миниатюр изображений и преобразования документов в формате TeX), а также изменять глобальные настройки (обратитесь к разделу "Конфигурация" выше).
- Зарегистрированные пользователи wiki из группы "sysops" (иногда также называемые "администраторы") могут редактировать гаджеты, настройки сценариев JavaScript и таблиц стилей CSS в рамках сайта.
- Любой зарегистрированный пользователь может настроить принцип работы приложения и пользовательский интерфейс по своему усмотрению (изменить существующие настройки, оболочки и гаджеты) или сделать свои собственные модификации (используя свои персональные настройки сценариев JS и таблиц стилей CSS на страницах).

Внешние программы также могут взаимодействовать с приложением MediaWiki посредством его системного API и, в случае его активации, сделать возможным доступ пользователя практически к любой функции и любым данным.

JavaScript и CSS

Приложение MediaWiki может читать и использовать сценарии JavaScript и таблицы стилей CSS на уровне сайта или оболочки с помощью специальных страниц wiki; эти страницы находятся в пространстве имен MediaWiki: и, таким образом, могут редактироваться только пользователями группы "sysops"; например, модификации сценария JavaScript с использованием страницы MediaWiki:Common.js повлияют на все оболочки, модификации таблицы стилей CSS с использованием страницы MediaWiki:Common.css также повлияют на все оболочки, но модификации таблицы стилей CSS с использованием страницы MediaWiki:Vector.css повлияют исключительно на пользователей оболочки с названием "Vector".

Пользователи могут совершать подобные модификации, которые будут затрагивать только используемый ими интерфейс путем редактирования подстраниц их пользовательской страницы (т.е., User:<Имя пользователя>/common.js для изменения сценария JavaScript всех оболочек,

User:<Имя пользователя>/common.css для изменения таблицы стилей CSS всех оболочек или User:<Имя пользователя>/vector.css для изменения таблицы стилей CSS оболочки под названием "Vector").

В том случае, если установлено расширение "Gadgets", пользователи из группы "sysops" могут также редактировать гаджеты, т.е., фрагменты кода на языке JavaScript, реализующие функции, которые могут быть включены и выключены пользователями с помощью их страницы настроек. Грядущие разработки в области гаджетов позволят различным разделам wiki совместно использовать гаджеты, что позволит избежать дублирования кода.

Этот набор инструментов имел большой успех и значительным образом повысил степень демократичности процесса разработки приложения MediaWiki. Отдельные разработчики имеют возможность самостоятельно добавлять функции; опытные пользователи могут делиться своими наработками с окружающими, обе группы делают это неофициально с помощью глобально настраиваемых систем, контролируемых пользователями из группы "sysops". Этот фреймворк идеален для небольших, не затрагивающих сторонние системы модификаций и предоставляет более низкий порог входления для разработчиков в отличие от сложных модификаций кода с использованием расширений и точек вызова функций.

Расширения и оболочки

Когда модификаций сценариев JavaScript и таблиц стилей CSS не достаточно, приложение MediaWiki предлагает использовать систему точек вызова функций, позволяющую сторонним разработчикам выполнять специфический код на языке PHP перед, после или вместо кода из состава приложения MediaWiki для обработки определенных событий. (Точки вызова функций приложения MediaWiki описаны в документе, расположенному по адресу <https://www.mediawiki.org/wiki/Manual:Hooks>.) Расширения приложения MediaWiki используют точки вызова функций для внедрения в код.

До того момента, как точки вызова функций были реализованы в приложении MediaWiki, добавление специфического кода на языке PHP подразумевало модификацию основного кода приложения, что было не так просто и не рекомендовалось делать. Первые точки вызова функций были предложены и реализованы Evan Prodromou в 2004 году; большее количество дополнительных точек вызова функций добавлялось при появлении необходимости в них в течение многих лет. С помощью точек вызова функций возможно добавить даже дополнительные возможности в язык разметки приложения MediaWiki, создав расширения тэгов.

Система расширений не идеальна; регистрация расширений осуществляется в ходе выполнения кода при запуске вместо использования кэшированных данных, что ограничивает возможности реализации абстракций и оптимизаций, а также негативно влияет на производительность приложения MediaWiki. Но, в общем, архитектура расширений в данный момент позволила реализовать относительно гибкую инфраструктуру, которая облегчила задачу по вынесению специализированного кода в модули, сдерживанию (высоких) темпов роста объема кода основных систем и добавлению сторонними пользователями функций в приложение MediaWiki.

Наоборот, очень сложно разработать новую оболочку для приложения MediaWiki без повторного изобретения колеса. В приложении MediaWiki оболочки являются классами языка PHP, каждый из которых расширяет возможности родительского класса Skin; они содержат функции, которые получают необходимую для генерации документа в формате HTML информацию. Существующая долгое время оболочка "MonoBook" сложно модифицируема из-за того, что она содержит большое количество специфического для браузеров кода таблиц стилей CSS, используемого для поддержки устаревших браузеров; редактирование шаблона или таблиц стилей CSS требовало множества последовательных изменений для поддержки совместимости со всеми браузерами и платформами.

API

Еще одной основной точкой входа для приложения MediaWiki, помимо файла `index.php`, является файл `api.php`, используемый в качестве API (интерфейса программирования приложений) для осуществления веб-запросов с использованием формата данных для программного доступа.

Пользователи сайта Wikipedia ранее создавали "ботов", которые получали данные путем обработки генерируемым приложением MediaWiki данных в формате HTML; этот метод был очень не надежным и много раз давал сбои. Для исправления ситуации разработчики представили интерфейс только для чтения данных (реализованный в файле `query.php`), который впоследствии был усовершенствован до полнофункционального поддерживающего программные операции чтения и записи данных API, предоставляющего прямой высокоуровневый доступ к информации из базы данных приложения MediaWiki. ([Объемная документация](#) для API доступна.)

Клиентские программы могут использовать API для входа в систему, получения данных и отправки изменений. API поддерживает как тонкие веб-клиенты на языке JavaScript, так и пользовательские приложения. Практически все действия, которые могут быть выполнены с помощью веб-интерфейса, также в общем случае могут выполнены с использованием API. Клиентские библиотеки, реализующие функции для доступа к API приложения MediaWiki доступны для многих языков программирования, включая языки Python и .NET.

12.9. Планы на будущее

Начатый единственным PHP-разработчиком в течение летних каникул проект увеличился в масштабе до приложения MediaWiki, отлаженной и стабильной системы wiki, под управлением которой работает находящийся в первой десятке по популярности вебсайт, использующий рабочую инфраструктуру удивительно малых масштабов. Это стало возможным благодаря постоянному процессу оптимизации с целью повышения производительности, последовательным архитектурным изменениям и команде замечательных разработчиков.

Эволюция веб-технологий и развитие проекта Wikipedia обусловили будущие усовершенствования и новые функции, некоторые из которых потребовали значительных изменений архитектуры приложения MediaWiki. Это хорошо прослеживается на примере реализации проекта нового визуального редактора, которая привела к восстановлению работы над системой разбора языка разметки и над самим языком разметки, его преобразованием в структуру DOM и финальным представлением в формате HTML.

MediaWiki является инструментом, используемым в очень различных целях. В рамках проектов организации Wikimedia, например, он используется для создания курирования энциклопедии (Wikipedia), для функционирования медиа-библиотеки большого объема (Wikimedia Commons), для перевода сканированных оригинальных текстов (Wikisource), и.т.д. В других условиях приложение MediaWiki используется в качестве корпоративной системы управления содержимым веб-сайта или в качестве репозитория данных, иногда в комбинации с семантическим фреймворком. Эти не запланированные при проектировании приложения специализированные способы эксплуатации, скорее всего, будут продолжать способствовать постоянным изменениям во внутренней структуре программного обеспечения. По существу, архитектура приложения MediaWiki является в значительной степени живой, точно также, как и поддерживающее ее развитие огромное сообщество пользователей.

12.10. Материалы для дополнительного чтения

- [Сайт документации и поддержки](#) приложения MediaWiki
- [Автоматически генерируемая документация](#) приложения MediaWiki

- Domas Mituzas, *Wikipedia: site internals, configuration, code examples and management issues*, MySQL Users conference, 2007. Полный текст доступен по адресу <http://dom.as/talks/>.

12.11. Благодарности

Эта глава была написана несколькими людьми совместно. Guillaume Paumier написал большую часть текста, используя информацию от пользователей и основных разработчиков приложения MediaWiki. Sumana Hariharan координировала проведение интервью и этапы сбора информации. Большая благодарность Antoine Musso, Brion Vibber, Chad Horohoe, Tim Starling, Roan Kattouw, Sam Reed, Siebrand Mazeland, Erik Moller, Magnus Manske, Rob Lanphier, Amir Aharoni, Federico Leva, Graham Pearce и другим людям за предоставление информации и/или рецензирование текста.

13. Moodle

[Moodle](#) является веб-приложением, обычно используемым в сфере образования. Хотя в данной главе и будет предпринята попытка создания обзора всех аспектов работы приложения Moodle, главным образом в главе будут рассматриваться те аспекты архитектуры Moodle, которые особенно интересны:

- Метод разделения кода между приложением и расширениями;
- Система прав доступа, которая контролирует то, какие пользователи какие действия могут выполнять в рамках различных частей системы;
- Способ генерации результирующего документа, предусматривающий использование различных тем (оболочек) для получения множества вариантов внешнего вида приложения, а также метод локализации интерфейса.
- Слой абстракции для работы с базой данных.

Приложение Moodle предоставляет сетевое пространство, в котором студенты и преподаватели могут объединиться для преподавания и получения знаний. Сайт Moodle разделен на курсы (*courses*). Курс содержит ассоциированный с ним список пользователей (*users*) с различными ролями, такими, как студент (*Student*) или преподаватель (*Teacher*). Каждый курс включает в себя некоторое количество ресурсов (*resources*) и действий (*activities*). Ресурс может быть представлен файлом в формате PDF, страницей в формате HTML в рамках приложения Moodle или ссылкой на какой-либо сетевой ресурс. Действие может быть представлено форумом, тестом или ресурсом wiki. В рамках курса эти ресурсы и действия структурированы каким-либо образом. Например, они могут быть сгруппированы на основе логически разделенных тем или календарных недель.

The screenshot shows a Moodle course page titled "Introduction to Moodle Programming". At the top, there is a navigation bar with links to "Home", "My courses", and "Moodle Programming". A "Turn editing on" button is also visible. Below the navigation bar, the title "Topic outline" is displayed. On the left side, there is a sidebar with icons and links: "Syllabus - Begin here!", "Course Schedule", "Questions? Ask them here ...", "Important terminology used throughout this course", "Announcements", and "Facilitator forum". The main content area shows a single section titled "1 Introduction and Preparation". Under this section, the "Objectives:" heading is listed with three bullet points: "Join the learning community by providing a self-introduction and greet classmates", "Differentiate between open source and closed source software", and "Join and explore the online Moodle community".

Рисунок 13.1: Курс Moodle

Moodle может использоваться в качестве отдельного приложения. В том случае, если вы пожелаете заняться преподаванием курсов по архитектуре программного обеспечения (например), вам

придется загрузить приложение Moodle на ваш веб-сервер, установить его, начать создавать курсы и ожидать студентов, которые должны самостоятельно зайти на сайт и зарегистрироваться. В качестве альтернативного решения, в том случае, если вы создаете курсы для учебного заведения значительного размера, Moodle может быть одной из систем, которые вы будете использовать. Возможно, у вас в распоряжении будет инфраструктура, изображенная на [Рисунке 13.2](#).

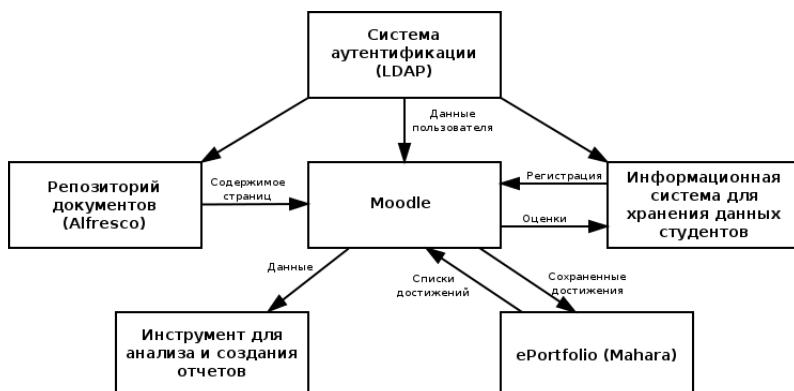


Рисунок 13.2: Типичная архитектура университетских систем

- Система аутентификации/идентификации (например, LDAP) для управления учетными записями пользователей всех ваших систем.
- Информационная система для хранения данных студентов; это база данных с информацией о всех ваших студентах, о том, по какой программе они обучаются и, следовательно, о курсах, которые им необходимо окончить; также с копиями этих данных - высокуюровневыми отчетами о результатах окончания курсов. Эта система также выполняет другие административные функции, такие, как слежение за тем, оплатили ли студенты курсы.
- Репозиторий документов (например, Alfresco); он используется для хранения файлов и отслеживания совместных действий пользователей по созданию файлов.
- ePortfolio; это место, в котором студенты могут собирать информацию или для создания CV (резюме) или для доказательства того, что они соответствуют требованиям курса, основанного на практических навыках.
- Инструмент для анализа и создания отчетов; используется для генерации высокуюровневой информации о том, что происходит в вашем учебном заведении.

Приложение Moodle в качестве основной функции предоставляет сетевое пространство для преподавания и получения знаний, в отличие от любой из других систем, которые могут потребоваться учебному заведению. Приложение Moodle предоставляет простейшую реализацию других функций, поэтому оно может функционировать или как отдельная система, или как система, интегрированная с другими системами. Приложение Moodle в общем случае выступает в роли, называемой "виртуальное образовательное окружение" (virtual learning environment - VLE) или "система управления процессом обучения или курсами" (learning or course management system - LMS, CMS или даже LCMS).

Приложение Moodle является программным обеспечением с открытым исходным кодом или свободным программным обеспечением (GPL). Оно разработано с использованием языка PHP. Приложение может функционировать на большинстве стандартных веб-серверов, на стандартных платформах. Оно требует наличия базы данных и работает с MySQL, PostgreSQL, Microsoft SQL Server или Oracle.

Проект Moodle был начат Martin Dougiamas в 1999 году, когда он работал в Университете Curtin в Австралии. Версия 1.0 была выпущена в 2002 году, в то время, когда PHP 4.2 и MySQL 3.23 являлись доступными технологиями. Это обстоятельство ограничило выбор возможного типа изначальной архитектуры, но впоследствии все значительно изменилось. Текущим релизом приложения Moodle является релиз из серии 2.2.x.

13.1. Обзор принципа работы приложения Moodle

Установка приложения Moodle состоит из трех составных частей:

1. Код, обычно расположенный в такой директории, как `/var/www/moodle` или `~/htdocs/moodle`. Доступ на запись в эту директорию не должен предоставляться веб-серверу.
2. База данных, управляемая одной из поддерживаемых систем управления реляционными базами данных (relational database management system - RDMS). На самом деле, приложение Moodle добавляет префикс ко всем именам таблиц, поэтому оно может делить базу данных с другими приложениями в случае необходимости.
3. Директория `moodledata`. Это директория, в которой приложение Moodle хранит загружаемые и генерируемые файлы, поэтому данная директория должна быть доступна для записи со стороны веб-сервера. В целях безопасности эта директория должна находиться вне корневой директории веб-сервера.

Все три части могут быть расположены на одном сервере. В качестве альтернативы при работе в окружении с системой балансировки нагрузки может использоваться множество копий кода, по одной на каждом из веб-серверов, но должна использоваться только одна разделяемая копия базы данных и одна директория `moodledata`, возможно на других серверах.

Конфигурационная информация для всех трех составных частей хранится в файле с названием `config.php` в корневой директории установки приложения Moodle с именем `moodle`.

Передача запросов

Moodle является веб-приложением, поэтому пользователи взаимодействуют с ним посредством своих веб-браузеров. С точки зрения приложения Moodle, процесс взаимодействия заключается в отправке ответов на HTTP-запросы. Следовательно, важным аспектом архитектуры Moodle является пространство имен URL и способ доставки данных URL различным сценариям.

Приложение Moodle в данном случае использует стандартный подход языка PHP. Стока URL для просмотра главной страницы курса будет иметь следующий вид: `.../course/view.php?id=123`, где 123 является уникальным идентификатором курса в базе данных. Стока URL для просмотра дискуссии форума будет выглядеть аналогично `.../mod/forum/discuss.php?id=456789`. То есть, эти сценарии `course/view.php` или `mod/forum/discuss.php` будут обрабатывать эти запросы.

Так проще для разработчика. Для понимания того, как приложение Moodle обрабатывает определенный запрос, следует рассмотреть строку URL и начать читать код в ней. Это неудачное решение с точки зрения пользователя. Эти строки URL, однако, постоянны. Строки URL не изменяются в случае переименования курса или в том случае, когда модератор перемещает дискуссию в другой форум. (Это свойство строк URL является полезным и описано в статье Tim Berners-Lee с названием "[Cool URIs don't change](#)".)

Альтернативным подходом, который может показаться полезным, является единая точка входа `.../index.php` [дополнительная-информация-для-превращения-запроса-в-уникальный]. После этого отдельный сценарий `index.php` будет предавать запросы каким-либо образом. Этот подход добавляет уровень абстракции, что всегда любят делать разработчики программного обеспечения. Отсутствие этого уровня абстракции не кажется вредным для приложения Moodle.

Расширения

Как и множество успешных проектов с открытым исходным кодом, проект Moodle построен на основе большого количества работающих совместно с ядром системы расширений. Этот подход удачен, так как он позволяет пользователям изменять и расширять возможности приложения Moodle различными способами. Важным преимуществом системы с открытым исходным кодом является тот факт, что вы можете адаптировать ее к вашим определенным потребностям. Выполнение значительных изменений кода может, однако, привести к большим проблемам при наступлении времени обновления, даже в случае использования удобной системы контроля версий. Позволяя производить так много изменений и добавлять столько новых функций, сколько возможно при условии их реализации в рамках самостоятельных расширений, взаимодействующих с ядром приложения Moodle с помощью описанного API, можно облегчить задачу по изменению возмож-

ностей Moodle пользователями в соответствии с их потребностями, а также по распространению внесенных изменений, при этом сохраняя возможность обновления ядра системы Moodle.

Существуют различные пути создания системы в виде ядра, окруженного расширениями. Приложение Moodle использует ядро относительно большого размера, при этом расширения являются строго типизированными. При разговоре о большом размере ядра я имею в виду большое количество функций, реализованных в рамках ядра. Этот подход противоположен используемому в архитектуре, где практически все функции, кроме небольшого загрузчика расширений, реализованы в рамках расширений.

Когда я говорю о строго типизированных расширениях, я имею в виду то, что вам придется разрабатывать различные типы расширений и использовать различные API в зависимости от того, какой тип функций вы хотите реализовать. Например, новое расширение в виде модуля действий будет значительно отличаться от расширения аутентификации или нового расширения типа вопроса. (Существует [полный список типов расширений приложения Moodle](#).) Этот подход противоположен используемому в архитектуре, где все расширения используют в основном один и тот же API а затем, возможно, соединяются с необходимым им подмножеством точек вызова функций или событий.

В основном, направление развития приложение Moodle предусматривало попытки сокращения размера ядра и переноса большего количества функций в расширения. Эти попытки были в какой-то степени успешны, однако, ввиду расширения набора функций сохранялась тенденция роста размера ядра. В рамках другого направления развития были предприняты попытки стандартизации различных типов расширений настолько, насколько это возможно, так, чтобы такие стандартные функции, как установки и обновление выполнялись аналогично во всех типах расширений.

Расширение приложения Moodle представлено в форме директории, содержащей файлы. Расширение имеет тип и название, которые совместно формируют составное название расширения, известное, как "Frankenstyle". (Слово "Frankenstein" появилось в ходе дискуссии на Jabber-канале разработчиков, при этом оно очень понравилось всем и его начали использовать.) Тип расширения и его название задают путь к директории расширения. Тип расширения задает префикс, а имя директории является названием расширения. Ниже приведено несколько примеров:

Тип расширения	Название расширения	Frankenstyle	Директория
mod (модуль действия)	forum	mod_forum	mod/forum
mod (модуль действия)	quiz	mod_quiz	mod/quiz
block (блок боковой панели)	navigation	block_navigation	blocks_navigation
qtype (тип вопроса)	shortanswer	qtype_shortanswer	question/type/shortanswer
quiz (отчет о тестировании)	statistics	quiz_statistics	mod/quiz/report/statistics

Последний пример иллюстрирует то, что каждый модуль действия может объявлять типы подмодулей. На данный момент только модули действий могут делать это по двум причинам. Если бы всем расширениям было позволено иметь подрасширения, могли бы возникнуть проблемы с производительностью. Модули действий реализуют основные обучающие возможности приложения Moodle и поэтому являются самым важным типом расширений, следовательно, они имеют особые привилегии.

Пример расширения

Я смогу объяснить множество подробностей реализации архитектуры приложения Moodle, рассмотрев пример специального расширения. По традиции я решил реализовать расширение, которое будет выводить фразу "Hello world".

Это расширение на самом деле полностью не соответствует ни одному из стандартных типов расширений приложения Moodle. Это просто сценарий, не связанный с чем-либо еще, поэтому я решил остановить свой выбор на реализации "локального" ("local") расширения. Этот тип расширения является обобщенным типом для реализации разнообразных функций, которые не могут быть корректно отнесены к какому-либо из других типов. Я назову свое расширение `greet` для формирования названия Frankenstyle `local_greet` и пути к директории `local/greet`. (Исходный код расширения [может быть загружен](#).)

Каждое расширение должно содержать файл с названием `version.php`, который содержит некоторые основные метаданные, относящиеся к расширению. Этот файл используется системой установки расширений приложения Moodle для установки и обновления расширения. Например, файл `local/greet/version.php` содержит строки:

```
<?php
$plugin->component      = 'local_greet';
$plugin->version         = 2011102900;
$plugin->requires        = 2011102700;
$plugin->maturity        = MATURITY_STABLE;
```

Включение названия компонента в файл может показаться избыточным, так как оно может быть извлечено из пути к директории, но установщик использует его для проверки того, что расширение установлено в предназначенному для него месте. Поле версии содержит версию расширения. Поле `maturity` содержит указание на альфа-версию (ALPHA), бета-версию (BETA), релиз-кандидат (RC, release candidate) или стабильную версию (STABLE). Поле `requires` указывает на минимальную версию приложения Moodle, с которой совместимо данное расширение. Если это необходимо, есть возможность указать другие расширения, от которых зависит данное расширение.

Ниже приведен код главного сценария для этого простого расширения (расположенный в файле

```
<?php
require_once(dirname(__FILE__) . '/../../config.php'); // 1

require_login(); // 2
$context = context_system::instance();
require_capability('local/greet:begreeted', $context);

$name = optional_param('name', '', PARAM_TEXT);
if (!$name) {
    $name = fullname($USER); // 3
}

local/greet/index.php): // 4
```

Строка 1: Начальная загрузка приложения Moodle

```
require_once(dirname(__FILE__) . '/../../config.php'); // 1
```

Единственной строкой этого сценария, выполняющей большую часть работы, является первая строка. Выше я упоминал о том, что файл `config.php` содержит данные, которые требуются приложению Moodle для соединения с базой данных и поиска директории с данными приложения Moodle. Этот файл, однако, заканчивается строкой `require_once('lib/setup.php')`. При использовании этого файла:

1. загружаются все стандартные библиотеки приложения Moodle с помощью функции `require_once`;
2. инициируется запуск системы обработки сессий;
3. осуществляется соединение с базой данных; и
4. устанавливаются значения большого количества глобальных переменных, с которыми мы столкнемся позднее.

Строка 2: Установление факта входа пользователя в систему

```
require_login(); // 2
```

Эта строка вынуждает приложение Moodle проверить, осуществил ли текущий пользователь вход в систему, используя любое из расширений аутентификации, настроенных администратором. Если пользователь не осуществил вход, он будет перемещен на страницу с формой ввода данных для входа в систему и эта функция никогда не вернет управление.

Сценарий, интегрированный с приложением Moodle лучшим образом, передаст большее количество аргументов для того, чтобы, скажем, установить, частью какого курса или действия является эта страница, а после этого функция `require_login` проверит, зарегистрирован ли пользователь в системе или наоборот, разрешен ли пользователю доступ к этому курсу и просмотр этого действия. В противном случае будет выведено соответствующее сообщение об ошибке.

13.2. Система ролей и разрешений приложения Moodle

Следующие две строки кода демонстрируют метод проверки того, что у пользователя есть разрешения на выполнение некоторых действий. Как вы можете видеть, с точки зрения разработчика, API является очень простым. На заднем плане, однако, находится усложненная система прав доступа, которая позволяет администратору осуществлять гибкий контроль над тем, что и кто имеет право делать.

Строка 3: Получение контекста

```
$context = context_system::instance(); // 3
```

В приложении Moodle пользователи могут иметь различные права в различных местах. Например, пользователь может быть преподавателем на одном курсе и студентом на другом и, таким образом, иметь различные права в каждом из мест. Эти места называются контекстами (*contexts*). Контексты в приложении Moodle формируют иерархию, очень похожую на иерархию директорий в файловой системе. На верхнем уровне находится контекст системы (System context) (и, так как данный сценарий не достаточно хорошо интегрирован с приложением Moodle, он использует именно этот контекст).

В системном контексте находится некоторое количество контекстов для различных категорий, которые создаются для организации курсов. Они могут быть вложенными, поэтому категория может содержать другие категории. Контексты категорий (Category contexts) также могут содержать контексты курсов (Course contexts). Наконец, каждое действие в рамках курса имеет свой контекст модуля (Module context).

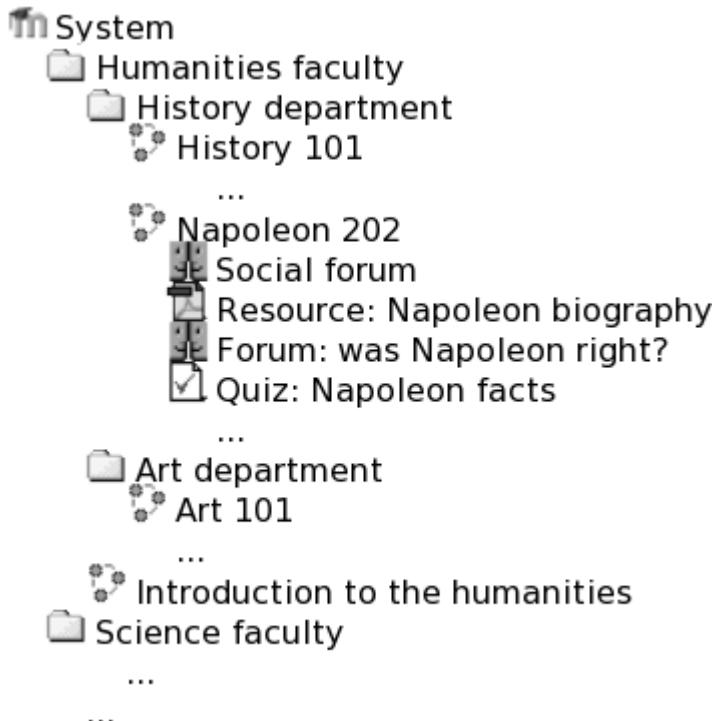


Рисунок 13.3: Контексты

Строка 4: Проверка того, имеет ли пользователь разрешение на использование данного сценария

```
require_capability('local/greet:begreeted', $context); // 4
```

При наличии контекста - соответствующей части приложения Moodle - разрешение может быть проверено. Каждый набор функций, который предоставляется или не предоставляется пользователю, называется возможностью (*capability*). Проверка возможностей позволяет реализовать более точный контроль доступа, чем базовые проверки, выполняемые с помощью функции `require_login`. Наш пример расширения имеет только одну возможность:

`local/greet:begreeted`.

Проверка осуществляется с помощью функции `require_capability`, которая принимает идентификатор возможности и контекст. Как и другие функции с префиксом `require_...`, она не вернет управление в том случае, если пользователь не имеет заданной возможности. Вместо этого она выведет ошибку. В других местах может применяться не фатальная функция `has_capability`, которая возвращает логическое значение, например, для установления того, показать ли ссылку на этот сценарий на другой странице.

Как администратор устанавливает то, какой пользователь имеет какие возможности? Ниже приведен список действий, которые выполняет функция `has_capability` (по крайней мере, концептуально):

1. Начинается работа с действующим контекстом.
2. Извлекается список ролей, которыми пользователь обладает в рамках данного контекста.
3. После этого проверяется то, какими разрешениями обладает пользователь в каждой роли в рамках данного контекста.
4. Эти разрешения объединяются для формирования финального ответа.

Описание возможностей

Как видно из примера, расширение может описывать новые возможности, соответствующие предоставляемым расширением функциям. В каждом расширении приложения Moodle присутствует

поддиректория для кода с именем db. Она содержит всю информацию, требующуюся для установки или обновления расширения. Одним из файлов, хранящих эту информацию, является файл с именем access.php, который описывает возможности. Ниже приведено содержимое файла access.php из состава нашего расширения, который находится в директории local/greet/db/access.php:

```
<?php
$capabilities = array('local/greet:begreeted' => array(
    'captopype' => 'read',
    'contextlevel' => CONTEXT_SYSTEM,
    'archetypes' => array('guest' => CAP_ALLOW, 'user' => CAP_ALLOW)
));

```

Здесь представлены некоторые относящиеся к данной возможности метаданные, которые используются при формировании пользовательского интерфейса для управления разрешениями. Также они используются для установления начальных разрешений для стандартных типов ролей.

Роли

Следующим элементом системы разрешений приложения Moodle являются роли. Роль (*role*) на самом деле представляет из себя именованный набор разрешений. После того, как вы осуществили вход в систему Moodle, вы получите роль "автентифицированного пользователя" в рамках системного контекста, а так как системный контекст является корневым в иерархии контекстов, эта роль будет применима везде.

В рамках определенного курса вы можете быть студентом и это соответствие роли будет актуально для контекста курса и всех контекстов модулей в нем. В другом курсе, однако, вы можете выступать в другой роли. Например, Mr Gradgrind может быть преподавателем курса "Facts, Facts, Facts", но при этом быть студентом курса профессиональной разработки "Facts Aren't Everything". Наконец, пользователю может быть предоставлена роль модератора в одном определенном форуме (в рамках контекста модуля).

Разрешения

Роль описывает разрешение (*permission*) для каждой возможности. Например, роль преподавателя наверняка установит значение разрешения ALLOW (позволяющее использовать) возможность moodle/course:manage, а роль студента - нет. Однако, и студенту и преподавателю будет доступна возможность mod/forum:startdiscussion.

Роли обычно описываются глобально, но они могут быть описаны повторно в каждом из контекстов. Например, определенный ресурс wiki может быть открыт только для чтения для студентов с помощью изменения значения разрешения для возможности mod/wiki:edit роли студента в рамках контекста этой системы wiki (контекста модуля) на значение PREVENT (предотвращающее использование).

Существуют четыре значения разрешения:

- NOT SET/INHERIT (неустановленное/наследуемое значение, являющееся стандартным)
- ALLOW (разрешающее значение)
- PREVENT (запрещающее значение)
- PROHIBIT (запрещающее, не изменяемое в подконтекстах значение)

В заданном контексте роль будет иметь одно из этих четырех значений разрешений для каждой из возможностей. Единственным различием между значениями PREVENT и PROHIBIT является то, что значение PROHIBIT не может быть изменено в подконтекстах.

Объединение разрешений

В конечном счете разрешения для всех ролей, в которых пользователь выступает в активном контексте, объединяются.

- Если любая роль использует значение разрешения PROHIBIT для возможности, возвращается логическое ложное значение (false).
- В ином случае, если любая роль использует значение разрешения ALLOW для возможности, возвращается логическое истинное значение (true).
- В других случаях возвращается логическое ложное значение (false).

Условия использования значения PROHIBIT следующие: представьте, что пользователь создает оскорбительные сообщения на множестве форумов и нам нужно немедленно прекратить это. Мы можем создать роль Naughty, в рамках которой установить значение PROHIBIT для возможности mod/forum:post и других аналогичных возможностей. После этого мы присваиваем эту роль нарушающему порядок пользователю в системном контексте. Таким образом мы можем удостовериться в том, что этот пользователь не сможет написать ни одного сообщения ни в одном из форумов. (После этого мы свяжемся со студентом и в том случае, если получим от него удовлетворяющий нас ответ, сможем удалить соответствие этой роли, после чего он вновь сможет использовать систему.)

Таким образом, система разрешений приложения Moodle позволяет администраторам осуществлять очень гибкое управление системой. Они могут описывать желаемые роли с различными разрешениями для каждой из возможностей; они могут изменять заданные значения в рамках ролей в подконтекстах; а также они могут ставить в соответствие пользователям в различных контекстах различные роли.

13.3. Вернемся к нашему примеру сценария

Следующая часть сценария иллюстрирует некоторые дополнительные функции:

Строка 5: Получение данных запроса

```
$name = optional_param('name', '', PARAM_TEXT); // 5
```

Действием, которое должно выполнять любое веб-приложение, является получение данных запроса (переменных, передаваемых с помощью методов запросов GET или POST) с необходимыми преобразованиями для устранения уязвимости приложения к атакам на основе SQL-инъекций или межсайтового скрипtingа. Приложение Moodle предоставляет два метода выполнения этого действия.

Простой метод показан выше. При его использовании извлекается значение отдельной переменной на основании имени параметра (в данном случае name), значения по умолчанию и ожидаемого типа данных. Ожидаемый тип данных используется для удаления из входящих данных всех не соответствующих ему символов. Существует множество типов, таких, как PARAM_INT, PARAM_ALPHANUM, PARAM_EMAIL и т.д.

Существует также функция, аналогичная функции required_param, которая также, как и все функции семейства require_... прекращает выполнение сценария и выводит сообщение об ошибке в том случае, если ожидаемый параметр не обнаружен.

Другим механизмом, поддерживаемым приложением Moodle и предназначенным для получения данных запроса является использование полнофункциональной библиотеки для работы с формами. Этот способ реализуется благодаря коду совместимости с библиотекой HTML QuickForm из

PEAR. (Для разработчиков, не имеющих опыта работы с языком PHP, следует отметить, что PEAR является эквивалентом хранилища CPAN для языка PHP.) Эта библиотека казалась хорошим выбором во время поиска подходящего решения, но ее поддержка на сегодняшний день не осуществляется. В какой-то момент в будущем нам придется перейти к использованию новой библиотеки для работы с формами, чего ожидают многие из нас, так как библиотека QuickForm имеет несколько раздражающий архитектурных недостатков. На сегодняшний день, однако, она соответствует предъявляемым требованиям. Формы могут быть описаны как коллекция полей различных типов (т.е. текстовых полей, выпадающих списков для выбора элементов, полей для выбора дат) с проверкой данных на стороне клиента и сервера (включая использование описанных типов, использующих префикс PARAM_...).

Строка 6: Глобальные переменные

```
if (!$name) {
    $name = fullname($USER);                                // 6
}
```

Этот фрагмент кода иллюстрирует первую глобальную переменную из набора глобальных переменных приложения Moodle. Переменная \$USER позволяет получить информацию о пользователе, получившим доступ к данному сценарию. Другие глобальные переменные:

- \$CFG: содержит часто используемые переменные конфигурации.
- \$DB: соединение с базой данных.
- \$SESSION: слой совместимости с сессией PHP.
- \$COURSE: курс, к которому относится данный запрос.

а также некоторые другие переменные, с некоторыми из которых мы будем иметь дело ниже.

Вы могли прочитать слова "глобальная переменная" с ужасом. Однако, следует заметить, что язык PHP обрабатывает единственный запрос в каждый момент времени. Следовательно, эти переменные не настолько глобальны. Фактически глобальные переменные языка PHP могут рассматриваться как реализация шаблона проектирования реестра уровня потока (обратитесь к книге Martin Fowler *Patterns of Enterprise Application Architecture*) и это тот метод, в соответствии с которым приложение Moodle использует эти переменные. Очень удобной чертой этого подхода является тот факт, что он делает доступными часто используемые объекты в коде без необходимости передачи их каждой функции и методу. Использованием этого подхода крайне редко злоупотребляют.

Все не так просто

Эта строка также позволяет описать проблемную область кода: не все так просто. Для вывода имени пользователя нужно применить более сложный подход, чем простое объединение строк \$USER->firstname, ' ' и \$USER->lastname. В школе может действовать распоряжение о выводе любой из этих частей, а также различные культуры используют различные соглашения о том, в каком порядке выводить составные части имени пользователя. Следовательно, существует ряд переменных конфигурации и функция для объединения частей имени в соответствии с этими правилами.

Та же проблема актуальна и для дат. Различные пользователи могут находиться в различных часовых поясах. Приложение Moodle хранит все даты в формате меток времени Unix, которые являются целочисленными значениями и, следовательно, совместимы со всеми базами данных. Существует функция userdate для вывода метки времени пользователю, использующему соответствующие настройки часового пояса и локализации.

Строка 7: Журналирование

```
add_to_log(SITEID, 'local_greet', 'begreeted',
           'local/greet/index.php?name=' . urlencode($name)); // 7
```

Все важные операции в рамках приложения Moodle подлежат журналированию. Журналы событий сохраняются в таблице базы данных. Это компромисс. Этот подход позволяет производить подробный анализ журнала достаточно просто, к тому же различные отчеты, сформированные на основе журналов событий, включены в комплект поставки приложения Moodle. Однако на сайте большого размера с большим количеством посетителей этот подход приводит к проблемам с производительностью. Таблица журнала событий увеличивается в размере, затрудняет резервное копирование базы данных и замедляет выполнение запросов данных. Также при записи данные в таблице могут объединяться. Эти проблемы могут быть преодолены различными путями, например, с помощью объединения операций записи или путем архивирования или удаления устаревших записей для перемещения их из основной базы данных.

13.4. Генерация результирующего документа

Процесс генерации результирующего документа в большей степени зависит от двух глобальных объектов.

Строка 8: Глобальная переменная \$PAGE

```
$PAGE->set_context($context); // 8
```

Переменная \$PAGE хранит информацию о странице, которая должна быть сгенерирована. Эта информации впоследствии доступна для кода, генерирующего результирующий HTML-документ. Для работы этого сценария требуется явное указание используемого контекста. (В других случаях контекст может быть установлен автоматически с помощью функции require_login.) Страна URL для этой страницы также должна быть явно задана. Это может показаться избыточным, но обоснование требования этих данных заключается в том, что вы можете перейти на определенную страницу, используя любое количество различных строк URL, но строка URL, передаваемая функции set_url должна быть канонической строкой URL для страницы - качественной постоянной ссылкой, если вам угодно. Заголовок страницы также устанавливается. Этот процесс заканчивается генерацией элемента head документа HTML.

Строка 9: Страна URL приложения Moodle

```
$PAGE->set_url(new moodle_url('/local/greet/index.php'),
                array('name' => $name)); // 9
```

Я хотел бы обратить внимание на этот замечательный небольшой вспомогательный класс, который позволяет значительно упростить манипуляции со строками URL. Отдельно следует вспомнить о том, что функция add_to_log, вызванная выше, не использует этот вспомогательный класс. В самом деле, API журналирования не может принимать объекты moodle_url. Этот тип несовместимости является типичным признаком значительного возраста кодовой базы приложения, такого, как у кодовой базы приложения Moodle.

Строка 10: Интернационализация

```
$PAGE->set_title(get_string('welcome', 'local_greet')); // 10
```

Приложение Moodle использует свою собственную систему для перевода интерфейса на любой из языков. На сегодняшний день может быть доступно большое количество библиотек интернационализации для языка PHP, но в 2002 году, когда приложение было впервые реализовано, не было доступно ни одной подходящей библиотеки. Система формируется вокруг функции `get_string`. Строки идентифицируются с помощью ключа и названия расширения в формате Frankenstyle. Как можно увидеть в строке 12, возможно преобразовать значения в строку. (Множества значений обрабатываются с использованием массивов объектов языка PHP.)

Поиск строк осуществляется в языковых файлах, содержащих простые массивы языка PHP. Ниже приведен языковой файл `local/greet/lang/en/local_greet.php` для нашего расширения:

```
<?php
$string['greet:begreeted'] = 'Be greeted by the hello world example';
$string['welcome'] = 'Welcome';
$string['greet'] = 'Hello, {$a}!';
$string['pluginname'] = 'Hello world example';
```

Следует отметить то, что наряду с двумя строками, используемыми в нашем сценарии, в файле также присутствуют строки для указания названия возможности и названия расширения, которые отображаются в пользовательском интерфейсе.

Различные языки идентифицируются с помощью двухбуквенных кодов стран (в данном случае `en`). Языковые пакеты могут происходить от других языковых пакетов. Например, `fr_ca` (французский канадский) языковой пакет декларирует `fr` (французский) в качестве родительского языка и, таким образом, задает только те строки, которые отличаются от строк французского языка. Так как разработка приложения Moodle началась в Австралии, код `en` относится к британскому английскому языку, а языковой пакет `en_us` (американский английский) происходит от него.

Снова следует упомянуть о том, что простой API на основе функции `get_string` для разработчиков расширений скрывает большую часть сложностей, включая установление языка (который может зависеть от действующих пользовательских настроек или настроек определенного курса, который изучается в данный момент) и поиск среди всех языковых пакетов и родительских языковых пакетов с целью получения строки.

Создание файлов языковых пакетов и координация переводов осуществляется с помощью ресурса <http://lang.moodle.org/>, который использует приложение Moodle со специальным расширением ([local_amos](#)). Этот ресурс использует систему контроля версий Git и базу данных в качестве системы для хранения языковых файлов вместе с полной историей версий.

Строка 11: Начало операции вывода страницы

```
echo $OUTPUT->header(); // 11
```

Это еще одна безобидно выглядящая строка, которая делает гораздо больше, чем кажется. Дело в том, что перед выполнением любой операции вывода страницы должна быть выбрана подходящая тема (оболочка). Этот процесс может зависеть от комбинации контекста страницы и настроек пользователя. Значение переменной `$PAGE->context`, однако, было установлено в строке 8, поэтому глобальная переменная `$OUTPUT` не будет инициализирована в начале сценария. В общем случае, решение этой проблемы заключается в использовании особенности языка PHP для создания соответствующего объекта `$OUTPUT` на основе информации, получаемой из переменной `$PAGE` в первый момент вызова любого метода для вывода страницы.

Еще одной важной особенностью является то, что каждая страница приложения Moodle может содержать блоки (*blocks*). Эти элементы являются дополнительными настраиваемыми частями содержащимого страницы, которые обычно выводятся слева или справа от основного содержащего страницы. (Они являются своего рода расширениями.) Снова следует упомянуть о том, что определенный набор выводимых блоков зависит от гибко изменяемых параметров (которые могут контролироваться администратором) контекста страницы и других аспектов идентичности страницы. Следовательно, следующим шагом подготовки к выводу страницы является вызов функции `$PAGE->blocks->load_blocks()`.

Как только вся необходимая информация обработана, расширение тем (которое в полной степени контролирует внешний вид страницы) вызывается для генерации страницы, включая стандартные заголовочные и завершающие области страницы, если это необходимо. Этот вызов также отвечает за добавление данных блоков в необходимое место результирующего документа формата HTML. В середине выводимой страницы должен находиться тэг `div`, в котором будет находиться специфическое содержимое этой страницы. После генерации документа в формате HTML он разделяется на две части по тегу `div`, находящемуся перед началом основного содержащего. Первая часть возвращается, а вторая - сохраняется для последующего возврата с помощью функции `$OUTPUT->footer()`.

Строка 12: Операция вывода основной части страницы

```
echo $OUTPUT->box(get_string('greet', 'local_greet',
    format_string($name))); // 12
```

В этой строке производится вывод основной части страницы. В данном случае просто выводится сообщение приветствия. Приветствие, повторюсь, является локализованной строкой, в данном случае со значением, подставляемым в предназначено для него поле. Основной объект вывода `$OUTPUT` предоставляет множество таких полезных методов, как `box` для описания требуемых выводимых элементов с помощью высокогуровневых директив. Различные темы могут контролировать то, какой именно код HTML будет использован для вывода сообщения.

Данные, изначально полученные из переменной пользователя (`$name`) выводятся после обработки с помощью функции `format_string`. Это еще одно мероприятие для защиты от XSS-атак. Эта функция также позволяет пользователю применять фильтры текста (другой тип расширения). Примером фильтра может быть фильтр LaTeX, который заменяет такие входные данные, как `$$x + 1$$` на изображение математического действия. Я упомяну, но не буду подробно описывать то, что на самом деле существуют три различных функции (`s`, `format_string` и `format_text`), применимые в зависимости от определенного выводимого типа содержащего страницы.

Строка 13: Завершение операции вывода страницы

```
echo $OUTPUT->footer(); // 13
```

Наконец осуществляется вывод завершающей части страницы. Этот пример не иллюстрирует данную операцию, но приложение Moodle отслеживает все сценарии на языке JavaScript, необходимые для функционирования страницы, и выводит все необходимые тэги для подключения сценариев в завершающей части страницы. Это стандартная и разумная практика. Она позволяет пользователям видеть страницу без ожидания загрузки всех сценариев на языке JavaScript. Разработчик должен подключать сценарии на языке JavaScript с помощью таких вызовов API, как `$PAGE->requires->js('/local/greet/cooleffect.js')`.

Приводит ли эта практика к смешению логики и выводимых данных

Очевидно, что размещение выводимого кода в файле сценария `index.php` даже при высоком уровне абстракции ограничивает гибкость управления выводимыми данными с помощью тем. Это является еще одним признаком значительного возраста кодовой базы приложения Moodle. Глобальная переменная `$OUTPUT` была представлена в 2010 году как переходный этап в рамках мероприятий по отказу от устаревшего кода, в котором функции вывода и управления содержимым страницы были размещены в одном файле и перехода к архитектуре, в которой весь код вывода страницы был корректно отделен. Это также иллюстрирует неудачный способ генерации страницы, ее разделения на две части, после чего выводимые сценарием данные могут быть размещены между заголовочной и завершающей областью страницы. Как только код вывода страницы был выделен из описанного сценария и перенесен в сценарий, который называется сценарием вывода страницы приложения Moodle, темы получили возможность осуществлять полную (или частичную) замену функций кода вывода страницы для рассматриваемого сценария.

Небольшой рефакторинг может позволить переместить код вывода страницы из сценария `index.php` в сценарий вывода страницы. Завершающие строки сценария `index.php` (строки с 11 по 13) изменятся на:

```
$output = $PAGE->get_renderer('local_greet');
echo $output->greeting_page($name);
```

и появится новый файл `local/greet/renderer.php`:

```
<?php
class local_greet_renderer extends plugin_renderer_base {
    public function greeting_page($name) {
        $output = '';
        $output .= $this->header();
        $output .= $this->box(get_string('greet', 'local_greet', $name));
        $output .= $this->footer();
        return $output;
    }
}
```

Если теме необходимо полностью изменить этот вывод, она может объявить подкласс этого класса, в котором будет повторно объявлен метод `greeting_page`. Функция `$PAGE->get_renderer()` устанавливает подходящий класс вывода страницы для ее вывода в зависимости от используемой темы. Следовательно, код вывода (показа) страницы полностью отделен от кода управления из файла `index.php` и проведено усовершенствование расширения с уровня использования устаревшего кода Moodle до уровня использования архитектуры MVC ("модель-представление-поведение" - "model-view-controller").

13.5. Абстракция для работы с базой данных

Сценарий "Hello world" был достаточно простым, поэтому у меня не было необходимости в получении доступа к базе данных, но несмотря на это, некоторые используемые библиотечные вызовы приложения Moodle осуществляли запросы к базе данных. Ниже я кратко опишу уровень абстракции для работы с базой данных приложения Moodle.

Приложение Moodle использовало библиотеку ADODB в качестве основы уровня абстракции для доступа к базе данных, но мы столкнулись с трудностями во время ее использования, кроме того дополнительный уровень кода библиотеки оказывал существенное влияние на производительность. Из-за этого в версии 2.0 приложения Moodle мы перешли к использованию нашего собственного уровня абстракции, являющегося тонкой прослойкой между различными библиотеками для работы с базами данных языка PHP.

Класс moodle_database

Сердцем библиотеки является класс `moodle_database`. Он описывает интерфейс, предоставляемый глобальной переменной `$DB`, который позволяет осуществлять доступ к соединению к базой данных. Типичный пример использования:

```
$course = $DB->get_record('course', array('id' => $courseid));
```

Этот вызов переводится на язык SQL следующим образом:

```
SELECT * FROM mdl_course WHERE id = $courseid;
```

и возвращает данные в форме обычного объекта языка PHP с общедоступными полями, поэтому вы можете получать доступ к ним: `$course->id`, `$course->fullname`, и.т.д.

Такие простые методы, как этот, используются для простых запросов и простых обновлений и добавлений данных в базу. Иногда необходимо выполнить более сложные SQL-запросы, например, для формирования отчетов. Для этого случая существуют методы выполнения произвольных SQL-запросов:

```
$courseswithactivitycounts = $DB->get_records_sql(
    'SELECT c.id, ' . $DB->sql_concat('shortname', "''", 'fullname') . ' AS
coursename,
    COUNT(1) AS activitycount
    FROM {course} c
    JOIN {course_modules} cm ON cm.course = c.id
    WHERE c.category = :categoryid
    GROUP BY c.id, c.shortname, c.fullname ORDER BY c.shortname, c.fullname',
    array('categoryid' => $category));
```

Некоторые аспекты, которые следует учесть в данном случае:

- Имена таблиц помещены в {}, таким образом библиотека может найти их и подставить имя таблицы в качестве префикса.
- Библиотека использует специальные символы для добавления значений в SQL-запрос. В некоторых случаях эта функция использует возможности текущего драйвера базы данных. В других случаях осуществляется замена символов и вставка их в SQL-запрос с помощью функций для работы со строками. Библиотека поддерживает как именованные специальные символы, (как показано выше) так и анонимные, обозначаемые с помощью символа ?.
- Для выполнения запросов всеми поддерживаемыми базами данных должен использоваться только набор безопасных и стандартных SQL-запросов. Например, вы можете увидеть, что я использовал ключевое слово AS для ссылок на столбцы, но не для ссылок на таблицы. Оба эти условия необходимы.
- Несмотря на это, возможны и такие ситуации, когда не будет доступен набор стандартных SQL-запросов, которые будут выполняться всеми поддерживаемыми базами данных; например, каждая база данных использует свой способ объединения строк. В этих случаях используются функции совместимости для генерации корректных SQL-запросов.

Описание структуры базы данных

Другой областью, в которой системы управления базами данных значительно отличаются, является синтаксис языка SQL для описания таблиц. Для преодоления этой проблемы каждое расширение приложения Moodle (и ядро Moodle) описывает требуемые таблицы базы данных в файле формата XML. Система установки приложения Moodle производит разбор файлов `install.xml` и использует информацию из них для создания требуемых таблиц и индексов. Существует встроенный в приложение Moodle инструмент разработчика под названием XMLDB, который может помочь в создании и редактировании этих установочных файлов.

Если структура базы данных должна быть изменена между двумя релизами приложения Moodle (или расширения) разработчик ответственен за написание кода (с использованием дополнительного объекта базы данных, предоставляющего методы языка описания данных DDL) для обновления структуры базы данных с сохранением пользовательских данных. Следовательно, приложение Moodle будет всегда самостоятельно обновляться от одного релиза к другому, упрощая операции обслуживания для администраторов.

Одним спорным моментом, обусловленным тем, что приложение Moodle начало свое развитие с использовании версии 3 MySQL, является то, что база данных приложения Moodle не использует внешние ключи. Такое положение вещей приводит к тому, что ошибки остаются не обнаруженными даже с учетом того, что современные базы данных позволяют обнаружить проблему. Сложностью является то, что люди используют приложение Moodle на своих сайтах без внешних ключей в течение многих лет, поэтому почти наверняка в базах присутствуют несвязанные данные. Добавление ключей на данный момент будет невозможным без очень сложной работы по очистке баз данных. Несмотря на это, с момента включения в состав приложения Moodle версии 1.7 системы XMLDB (в 2006 году!) файлы install.xml содержат описания внешних ключей, которые должны присутствовать в базе, и мы все еще надеемся когда-нибудь выполнить всю необходимую работу для того, чтобы добавлять эти ключи в ходе процесса установки.

13.6. Что не было описано

Я надеюсь, я что мне удалось представить неплохой обзор принципа работы приложения Moodle. Ввиду ограничения объема главы я исключил из рассмотрения несколько интересных тем, включая темы о том, как расширения аутентификации, регистрации и оценок позволяют приложению Moodle взаимодействовать с информационными системами для хранения данных студентов, а также об интересном способе хранения загружаемых файлов приложением Moodle на основе их содержимого. Информация, касающаяся этих и других аспектов архитектуры Moodle может быть найдена в [документации для разработчиков](#).

13.7. Выученные уроки

Один из интересных аспектов работы над проектом Moodle заключается в том, что он получил развитие в качестве исследовательского проекта. Moodle позволяет (но не заставляет) использовать подход [социальной конструктивистской педагогики](#). То есть, мы лучшим образом учимся создавая что-то, а также учимся друг у друга при работе в сообществе. Вопрос доктора философии Martin Dougiamas по поводу проекта заключался не в том, эффективна ли эта модель для обучения, а в том, эффективна ли эта модель для развития проекта с открытым исходным кодом. То есть, можем ли мы рассматривать проект Moodle как попытку изучения возможности создания и использования виртуального образовательного окружения (VLE), причем эта попытка заключалась в непосредственном создании и использовании приложения Moodle сообществом, в котором преподаватели, разработчики, администраторы и студенты преподают и учатся друг у друга? Я считаю эту модель удачной для проекта разработки программного продукта с открытым исходным кодом. Основным местом встречи разработчиков и пользователей для взаимного обучения являются дискуссии в разделе форумов проекта Moodle, а также раздел базы данных ошибок.

Возможно, наиболее важным результатом этого исследовательского проекта является вывод о том, что вы не должны бояться начинать разработку с реализации наиболее простого возможного решения в первую очередь. Например, в ранних версиях приложения Moodle имелось только несколько жестко заданных ролей, таких, как преподаватель, студент и администратор. Этого было достаточно в течение многих лет, но в итоге на ограничения было обращено внимание. Когда пришло время проектирования системы ролей для приложения Moodle версии 1.7, у сообщества был большой опыт использования Moodle, а также большое количество запросов функций, которые указывали на то, что нужно людям для более гибкого управления системой доступа. Все это

помогло в проектировании такой простой системы ролей, как это возможно, но при этом и такой сложной, как это необходимо. (Фактически первая версия системы ролей была чрезмерно сложной, поэтому впоследствии она была немного упрощена в версии 2.0 приложения Moodle.)

Если вы рассматриваете программирование как деятельность по решению задач, вам может показаться, что в первый раз для приложения Moodle была выбрана неподходящая архитектура, а позднее пришлось потерять много времени на ее корректировку. Я могу сказать, что такая точка зрения неконструктивна при попытке решения сложных задач, встречающихся в реальной жизни. Во время начала разработки приложения Moodle никто не располагал в достаточной степени знаниями о том, как спроектировать используемую на данный момент систему ролей. С точки зрения обучающегося, различные ступени развития, пройденные приложением Moodle до достижения актуальной архитектуры, были необходимы и неизбежны.

При таком подходе к разработке возможно изменение практически любого аспекта системной архитектуры, как только вы узнаете больше о ней. Мне кажется, приложение Moodle показывает, что это возможно. Например, мы нашли способ последовательного рефакторинга кода для перехода от устаревших сценариев к архитектуре MVC. Это требует усилий, но, кажется, что тогда, когда это необходимо, ресурсы для реализации этих изменений могут быть найдены в сообществах проектов с открытым исходным кодом. С точки зрения пользователя, система последовательно развивается с каждым релизом.

14. NGINX

Глава 14 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

nginx (произносится, как "engine x") - это web-сервер с открытым исходным кодом, написанный российским разработчиком Игорем Сысоевым. С момента опубликования в 2004 году nginx фокусировался на высокой производительности, высокоэффективных параллельных вычислениях и минимизации использования оперативной памяти. Такие дополняющие функции web-сервера возможности, как балансировка нагрузки, кэширование, контроль доступа, контроль пропускной способности и умение эффективно интегрироваться с различными приложениями позволили nginx стать хорошим выбором для web-сайтов с современной архитектурой. В настоящее время nginx занимает второе место среди самых популярных web-серверов с открытым исходным кодом.

14.1. Почему высокоеффективные параллельные вычисления так важны?

В настоящее время Интернет имеет настолько широкое и повсеместное распространение, что сложно себе представить его отсутствие каких-нибудь десять лет назад. Это результат взрывного развития от перехода по ссылкам между текстами с разметкой HTML, основанных на NCSA, а затем и на web-серверах Apache, до 2 миллиардов пользователей по всему миру, находящихся всегда на связи. С широким распространением постоянно подключенных к сети персональных компьютеров, мобильных устройств, а теперь и планшетных компьютеров, облик Интернета быстро меняется и целые экономики становятся цифровыми и проводными. Онлайн-сервисы стали более сложными с явным уклоном в сторону мгновенного доступа к постоянно обновляющейся информации и развлечениям. Вопросы обеспечения безопасного ведения бизнеса в Интернете также сильно изменились. Соответственно, web-сайты сейчас гораздо сложнее, чем раньше, и требуют больше инженерных решений для повышения надежности и масштабируемости.

Одной из самых больших проблем при выборе архитектуры web-сайта всегда был параллелизм. С момента появления web-сервисов необходимость в одновременном выполнении различных опера-

ций постоянно растёт. Необходимость одновременного обслуживания популярными web-сайтами сотен тысяч или даже миллионов пользователей - вовсе не редкость. Десятилетие назад главной причиной параллелизма была медленная скорость клиентов, использующих ADSL- или dial-up-соединения. В настоящее время параллелизм обусловлен сочетанием мобильных клиентов и новой архитектуры приложений, которые обычно основываются на поддержании постоянного подключения, что позволяет клиенту получать свежие новости, твиты, новостные ленты друзей и т.п. Ещё одним важным фактором, приведшим к увеличению важности параллелизма, стало изменение поведения современных браузеров, которые открывают от 4 до 6 соединений к web-сайту для ускорения загрузки страницы.

Чтобы проиллюстрировать проблему медленных клиентов представьте себе простой web-сервер на основе Apache, который генерирует относительно короткий ответ размером 100 КБ - web-страницу с текстом или изображением. Генерация страницы и ответ на запрос могут занять доли секунды, но занимают 10 секунд из-за скорости передачи клиенту, имеющему скорость доступа 80 Кбит/с (10 КБ/с). То есть, web-сервер будет сравнительно быстро подготавливать содержимое страницы в 100 КБ и затем оставаться занятым ещё 10 секунд из-за медленной передачи клиенту. А теперь представьте, что у вас есть 1000 подключённых клиентов, которые одновременно запросили получение аналогичных страниц. Если на каждого клиента выделять всего по 1 МБ оперативной памяти, то это привело бы к необходимости выделения 1000 МБ (около 1 ГБ) оперативной памяти для передачи 100 КБ информации. В реальности, типовой web-сервер на базе Apache обычно выделяет более 1 МБ на одно соединение, а эффективная скорость мобильных клиентов, к сожалению, десятки кбит/с. Ситуация с отправкой информации медленным клиентам может быть в какой-то степени улучшена путём увеличения буферов обмена сокетов ядра операционной системы, но это не универсальное решение и может иметь нежелательные побочные эффекты.

С постоянно поддерживаемыми соединениями проблема параллелизма при обработке становится ещё более выраженной, так как клиенты с целью уменьшения времени на подключение не разрывают ранее установленные соединения и web-сервер вынужден для каждого подключённого клиента резервировать определённый объём памяти.

Следовательно, чтобы справиться с возросшей нагрузкой, связанной с увеличивающейся аудиторией и, как следствие, требованиями к параллелизму, а такжеправляться с этим и далее, web-сайт должен основываться на ряде очень эффективных "строительных" блоков. Другие части этого уравнения, такие как аппаратная часть (ЦПУ, ОЗУ, НЖМД), пропускная способность сети, системное программное обеспечение и архитектура системы хранения, очень важны, но именно программное обеспечение web-сервера принимает и обрабатывает соединения клиентов. То есть, web-сервер должен быть способен к нелинейному масштабированию с ростом числа одновременных соединений и количества запросов в секунду.

Apache не подходит?

Apache - это доминирующий сейчас в Интернете web-сервер, берущий начало в 1990-х годах. Первоначально его архитектура соответствовала операционным системам и аппаратному обеспечению того времени, соответствовала и состоянию Интернета, где было принято для каждого web-сайта выделять физический сервер с единственным экземпляром Apache. К началу 2000-х годов стало очевидно, что модель с выделенным web-сервером не может удовлетворять растущие потребности web-сервисов. Несмотря на прочную основу для будущего развития, заложенную в Apache, его архитектура предусматривала запуск своей копии для каждого нового соединения, что не подходило для нелинейной масштабируемости web-сайта. В итоге Apache стал web-сервером общего назначения и сосредоточился на увеличении количества разнообразных функций и сторонних расширений для обеспечения универсальной применимости к практически любому виду web-разработки. Однако, за всё необходимо платить: столь богатый и универсальный инструментарий в рамках одной программы снижает её возможности к масштабированию из-за увеличенного использования ЦПУ и памяти на каждое соединение.

Таким образом, аппаратная часть сервера, его операционная система и сетевые ресурсы перестали быть основной проблемой для развития web-сайта, что привело web-разработчиков по всему миру к поиску более эффективных средств организации web-серверов. Около десяти лет назад Daniel Kegel, ведущий разработчик программного обеспечения, [объявил](#), что "настало время, когда web-серверы должны обрабатывать десять тысяч клиентских соединений одновременно" и предложил называть всё это облачными сервисами Интернета. Манифест "C10K" Кегеля (Daniel Kegel) обострил проблему оптимизации web-серверов для одновременной поддержки большого числа клиентских соединений и nginx при таком подходе оказался одним из лучших.

С целью преодоления проблемы C10K в 10'000 одновременных соединений в nginx была заложена другая архитектура, подразумевающая лучшую нелинейную масштабируемость, как по числу поддерживаемых одновременных соединений, так и по числу запросов в секунду. nginx основан на событийно-ориентированной модели (event-based), что позволяет ему не порождать новый процесс или поток для каждого запроса web-страницы, как это делает Apache. В результате возрастание нагрузки стало более равномерным, а использование ресурсов памяти и ЦПУ управляемым. nginx мог теперь обрабатывать десятки тысяч одновременных соединений на сервере с обычной аппаратной частью.

С выходом первой версии nginx стало понятно, что он должен применяться совместно с Apache для выдачи поддерживаемой nginx статической информации вроде HTML, CSS, JavaScript и изображений, что позволяло снизить нагрузку и время отклика серверов приложений на базе Apache. В ходе развития в nginx были добавлены интеграция с приложениями с помощью FastCGI, uwsgi или SCGI протоколов и системами распределённого кеширования в оперативной памяти, такими как memcached. Также были добавлены другие полезные функции, такие как обратный прокси-сервер с балансировкой и кешированием. Эти дополнительные возможности превратили nginx в эффективный набор инструментов для построения масштабируемой web-инфраструктуры.

В феврале 2012 года был представлен релиз Apache 2.4.x. Несмотря на добавление в эту версию Apache новых модулей с ядром многопоточной обработки и прокси-сервером, направленными на повышение масштабируемости и производительности, ещё прошло слишком мало времени, чтобы говорить о соизмеримых производительности, параллелизме обработки и экономном использовании ресурсов в сравнении с web-сервером, изначально построенным на событийно-ориентированной модели. Было бы очень приятно увидеть лучшую масштабируемость в новой версии сервера приложений Apache, хотя и не понятно, как это поможет устранить узкие места на серверной стороне в типовых web-конфигурациях nginx + Apache.

Есть ли ещё преимущества при использовании nginx?

Обработка большого количества одновременных запросов с обеспечением высокой производительности и эффективности всегда была ключевым преимуществом при внедрении nginx. Однако, есть и другие не менее интересные преимущества.

В последние несколько лет web-архитекторы восприняли идею удаления зависимостей и отделения инфраструктуры приложений от web-сервера. Однако, то, что раньше существовало в виде web-сайта, основанного на LAMP (Linux, Apache, MySQL, PHP или Perl), теперь может не только основываться на LEMP ("E" означает "nginx"), но и всё чаще встречается в виде web-сервера с интегрированной инфраструктурой или на том же наборе приложений и баз данных, но взаимодействующих на иных принципах.

nginx очень хорошо подходит для этого, так как обеспечивает весь необходимый функционал: снижение нагрузки при обработке одновременных запросов, снижение времени отклика, поддержка SSL (Secure Socket Layer), работа со статической информацией, сжатие и кеширование, управление отказом обслуживания соединений и запросов и даже HTTP-потоковое вещание мультимедийной информации - всё это делает nginx более эффективным для применения в качестве первого

принимающего запорсы web-сервера. Он также реализует непосредственную интеграцию с memcached/Redis или другими NoSQL-решениями для повышения производительности при одновременной обработке большого количества пользователей.

С появлением и широким распространением различных языков программирования и комплектов разработчика всё большее и большее количество компаний начинает их применять, что приводит к изменению способов разработки и внедрения. nginx стал одним из наиболее важных компонентов этих меняющихся парадигм и уже помог многим компаниям создать и развивать свои web-сервисы быстро и в рамках запланированных бюджетов.

Первая строка исходного кода nginx была написана в 2002 году. В 2004 году он был выпущен под лицензией "BSD 2-Clause License". С тех пор количество пользователей nginx постоянно растёт, предлагаются новые идеи, представляются отчёты об ошибках, формируются предложения и замечания - всё это вместе чрезвычайно полезно и выгодно для всего сообщества.

Исходный код nginx является оригинальным и был полностью написан "с нуля" на языке программирования Си. nginx был портирован на множество архитектур и операционных систем, включая Linux, FreeBSD, Solaris, Mac OS, AIX и Microsoft Windows. nginx основывается на своих собственных библиотеках и стандартных модулях, имеющих очень мало внешних зависимостей помимо библиотеки языка С, за исключением zlib, PCRE и OpenSSL, которые могут быть исключены во время сборки при необходимости или по лицензионным соображениям.

И ещё несколько слов о Windows-версии nginx. nginx работает в среде Windows. Причём Windows-версия nginx больше похожа на доказательство правильно выбранной концепции, а не на полнофункциональнуюportedную версию. Есть определённые ограничения nginx, связанные с архитектурой ядра Windows и отсутствием на текущее время хорошего взаимодействия с ним. Известные проблемы Windows-версии nginx это: значительно меньшее число поддерживаемых одновременных соединений, более низкая производительность, отсутствие кеширования и отсутствие управления полосой пропускания. Функционал будущих Windows-версий nginx будет более полно соответствовать основной версии.

14.2. Обзор архитектуры Nginx

Традиционные процессно- или потоко-ориентированные модели подразумевают при одновременном обслуживании соединений порождение нового процесса или потока на каждое соединение и его блокирование при сетевых операциях или операциях ввода/вывода. В зависимости от способа применения такие модели могут быть крайне неэффективными с точки зрения использования ЦПУ и оперативной памяти. Порождение нового процесса или потока влечёт за собой подготовку новой среды окружения с выделением в оперативной памяти кучи (heap), стека и нового контекста исполнения. ЦПУ тратит дополнительное время на создание этих объектов, что, в конечном итоге, может привести к снижению производительности из-за частого переключения контекста исполнения. Все перечисленные выше проблемы проявляются в web-серверах со старой архитектурой, таких как Apache. Это компромисс между богатым функционалом и оптимальным использованием ресурсов сервера.

С момента своего создания nginx задумывался в качестве специализированного инструмента, обеспечивающего достижение более высоких производительности, плотности и экономичности использования ресурсов сервера при одновременной поддержке возможности динамичного роста web-сайта, что потребовало применения другой модели. На самом деле вдохновение пришло из развитых событийно-ориентированных механизмов, развиваемых в различных операционных системах. В результате появилась модульная, событийно-управляемая, асинхронная, однопоточная и свободная от блокировок архитектура, которая и стала основой nginx.

nginx для мультиплексирования и извещения о событиях использует один главный процесс, а исполнение конкретных задач назначает отдельным подчинённым процессам. Обработка соединений выполняется в высокоэффективных циклах обработки событий, помещённых в потоки. Такие потоки называются "исполнители" (*workers*), а их количество ограничено. В рамках каждого "исполнителя" nginx может обрабатывать тысячи одновременных подключений и запросов в секунду.

Структура исходного кода

Исходный код "исполнителя" состоит из ядра и функциональных модулей. Ядро отвечает за поддержание работы цикла обработки событий и выполнение соответствующих модулей на каждом из этапов обработки запроса. Модули обеспечивают наибольшую функциональность на уровне представлений и уровне приложений. Модули считывают и записывают в сеть и на систему хранения, преобразуют данные, выполняют фильтрацию исходящего трафика, исполняют серверные операции и передают запросы вышестоящим серверам при включённом режиме проксирования.

Модульная архитектура nginx позволяет разработчикам расширять набор функций без изменения его ядра. Модули nginx подразделяются на несколько типов: модули ядра (*core modules*), обработчики событий (*event modules*), обработчики фазы (*phase handlers*), модули реализации протоколов (*protocols*), обработчики именованных переменных (*variable*), фильтры, модули перенаправления запросов на вышестоящие серверы (*upstreams*) и балансирующие нагрузки (*load balancers*). В настоящее время nginx не поддерживает динамически загружаемые модули. То есть, модули компилируются вместе с ядром в один исполняемый файл на этапе сборки. Однако, поддержка загружаемых модулей и ABI запланирована на следующие major-релизы. Более подробную информацию о роли различных модулей можно найти в разделе 14.4.

При обработке операций, связанных с приёмом, обработкой и управлением сетевыми подключениями, извлечением данных, а также при дисковых операциях ввода/вывода, nginx для повышения производительности использует специфические механизмы уведомления о событиях, такие как `kqueue`, `epoll` и `event ports`, доступные в операционных системах Linux, Solaris и семействе BSD. Цель состоит в максимальном использовании возможностей операционной системы для обеспечения своевременной обратной связи, по своей природе асинхронной, при обработке входящего/исходящего трафика, дисковых операций, чтения или записи в сокеты, организации тайм-аутов и т.п. Используемые в nginx различные методы мультиплексирования и ускоренного ввода/вывода в значительной степени оптимизированы для каждой Unix-подобной операционной системы, на которой он запускается.

Обзорная схема архитектуры nginx представлена на рисунке 14.1

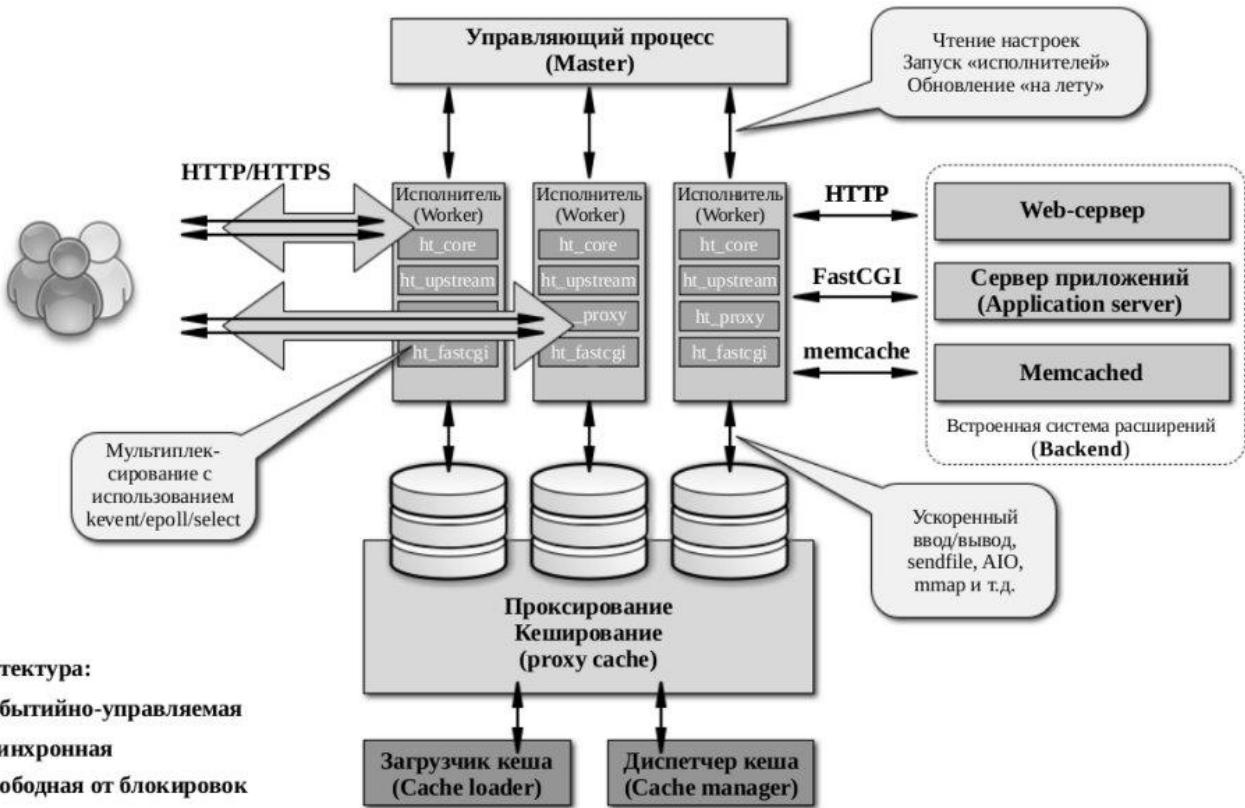


Рисунок 14.1: Схема архитектуры nginx

Модель исполнителя

Как уже упоминалось ранее, nginx не порождает новый процесс или поток для каждого соединения. Вместо этого исполнители принимают новые запросы из разделяемого прослушиваемого сокета и выполняют их в своих высокоеффективных циклах обработки, что позволяет обрабатывать тысячи соединений на каждого исполнителя. В nginx нет специализированных механизмов управления или перераспределения соединений между исполнителями, так как эта работа выполняется подсистемами ядра операционной системы. После запуска выполняется создание сокетов для приёма соединений. Затем исполнители непрерывно принимают, считывают и записывают в эти сокеты до тех пор пока выполняются обработка HTTP-запросов.

Непрерывный цикл обработки является самой сложной частью исходного кода исполнителя. Здесь широко используются внутренние вызовы и идея асинхронной обработки. Асинхронные операции реализованы с применением модульности, механизма сообщений о событиях, широкого использования функций обратного вызова (callback functions) и великолепно доработанных таймеров. В целом, во всём прослеживается принцип: все операции должны быть неблокирующими настолько, насколько это возможно. Единственная ситуация, когда nginx всё-таки применяет блокировку - это нехватка производительности дисковой подсистемы для исполнителя.

Так как nginx не порождает новый процесс или поток для каждого соединения, использование оперативной памяти очень экономичное и в подавляющем большинстве случаев крайне эффективное. nginx экономно использует ЦПУ ввиду отсутствия постоянных созданий и удалений контекстов процесса или потока. Всё что делает nginx, это проверяет состояние сетевого подключения и дисковой подсистемы, инициирует создание новых соединений, переносит их обработку в цикл исполнителя и асинхронно обрабатывает их до завершения, после чего соединение освобождается и удаляется из цикла обработки исполнителя. Сочетание бережного использования вызовов `syscall` с аккуратной реализацией интерфейсов распределителей памяти `pool` и `slab` (`pool and slab memory allocators`) позволяет nginx достигать от умеренной до низкой загрузки ЦПУ даже при экстремальных нагрузках.

В связи с тем, что nginx использует несколько исполнителей для обработки соединений, он хорошо масштабируется на несколько ядер ЦПУ. В целом, распределение исполнителей по ядрам позволяет полностью использовать многоядерные архитектуры и предотвращает простой потоков и блокировок. В итоге, отсутствует нехватка ресурсов, а механизмы их контроля изолированы каждый в своём потоке исполнителя. Такая модель также позволяет достичь лучшей масштабируемости дисковой подсистемы, способствует более эффективному её использованию и позволяет избежать блокировок при операциях ввода/вывода. В результате, ресурсы сервера используются более эффективно с распределением нагрузки между исполнителями.

Для достижения заданных паттернов использования дисковой подсистемы и ЦПУ число исполнителей может быть скорректировано. Для этого есть несколько основных правил, которые системные администраторы должны попробовать для своих рабочих нагрузок. Общие рекомендации могут быть следующими: если предполагается высокая нагрузка на ЦПУ (например, обработка большого количества TCP/IP запросов, обработка SSL или сжатие), то число исполнителей должно совпадать с количеством процессорных ядер; если предполагается высокая нагрузка на дисковую систему ввода/вывода (например, обслуживание запросов на выдачу данных или высоконагруженное проксирование), то число исполнителей должно быть в полтора-два раза больше, чем количество процессорных ядер. Некоторые инженеры выбирают количество исполнителей по количеству систем хранения или дисков, но эффективность такого подхода зависит от типа и конфигурации дисковой подсистемы.

Одной из основных проблем, которую разработчики nginx будут решать в ближайших версиях, является существенное уменьшение количества блокировок при дисковых операциях ввода/вывода. На данный момент, если одному из исполнителей не хватает производительности дисковой подсистемы, то он может не снять блокировку на дисковый ввод/вывод пока не завершит свою операцию. Однако, существует целый ряд механизмов и директив конфигурационного файла для смягчения сценариев с блокировкой дискового ввода/вывода. В частности, комбинация опций `sendfile` и `AIO` обычно обеспечивает достаточный запас по производительности дисковой подсистемы. То есть, развёртывание nginx должно планироваться с учётом типа информации, к которой будет обеспечиваться web-доступ, объёма доступной оперативной памяти и архитектуры системы хранения данных.

Другая проблема в существующей модели исполнителя - это ограниченная поддержка встраиваемых скриптовых сценариев. В стандартной поставке nginx поддерживается встраивание скриптов только на языке Perl. Этому есть простое объяснение: встроенный скрипт может заблокировать выполнение любой операции или неожиданно завершиться. Любой из этих двух вариантов поведения может привести к зависанию исполнителя, что отразится на тысячах одновременных соединений, которые он обслуживал. Планируется проделать большую работу, чтобы сделать встраивание скриптовых языков более простым, надёжным и пригодным для широкого спектра применений.

Назначение процессов nginx

nginx запускает несколько процессов в оперативной памяти: один главный процесс и несколько процессов исполнителей. Также есть несколько специализированных процессов, в частности, загрузчик кеша и диспетчер кеша. В nginx версии 1.1 все процессы однопоточные. Все процессы для взаимодействия между собой используют механизмы разделаемой памяти (*shared-memory mechanisms*). Управляющий процесс nginx (master process) запускается с привилегиями пользователя `root`. Загрузчик кеша, диспетчер кеша и исполнители запускаются от имени обычного непrivилегированного пользователя.

Управляющий процесс отвечает за выполнение следующих задач:

- считывание и проверка на корректность файла настроек

- создание, подключение и удаление сокетов
- запуск, завершение и поддержание заданного количества исполнителей
- загрузка новых настроек без остановки работы
- управление бинарными обновлениями "на лету" (начиная от замены исполняемого файла и заканчивая откатом к предыдущему состоянию при необходимости)
- повторные открытия лог-файлов
- трансляция встроенных скриптов на языке Perl

Процессы исполнителей принимают, управляют и обрабатывают соединения от клиентов, обеспечивают обратное проксирование и фильтрацию, то есть, делают почти всё, на что способен nginx. Что касается контроля за работой nginx, то системный администратор должен следить за состоянием процессов исполнителей, так как, фактически, именно они изо дня в день выполняют все операции web-сервера.

Процесс загрузчика кеша отвечает за контроль целостности кеша, хранящегося на диске, и заполнение базы данных с кешем мета-информации в памяти nginx. По сути, загрузчик кеша позволяет nginx работать с уже сохранёнными на жёсткий диск файлами, организованными в специальную структуру каталогов. При запуске он обходит все каталоги, проверяет целостность кеша мета-данных, обновляет соответствующие записи в разделяемой памяти и когда все устаревшие записи удалены, а остальные готовы к использованию, завершается.

Диспетчер кеша в основном отвечает за отслеживание истечения срока действия записей и маркирование их как недействительных. Он остаётся в оперативной памяти всё время работы nginx и в случае сбоя перезапускается вместе с главным процессом.

Краткий обзор кеширования в nginx

Кеширование в nginx реализовано в виде иерархически организованного хранилища в файловой системе. Ключи кеша настраиваемые, а информация, попадающая в кеш, может контролироваться с помощью различных специфичных для каждого типа запросов параметров. Кеш ключей и кеш мета-данных хранятся в сегменте разделяемой памяти, к которой могут получить доступ диспетчер кеша и исполнители. В настоящее время кеширование в оперативную память отсутствует, кроме возможной оптимизации с помощью механизма виртуальной файловой системы средствами операционных систем. Кеш для каждого запроса располагается в отдельном файле в файловой системе. Иерархией кеша (уровни и способы именования) можно управлять с помощью директив конфигурационного файла nginx. При записи запроса в структуру каталогов кеша путь и имя файла вычисляются с помощью MD5-хеша проксируемого адреса URL.

Процесс размещения данных в кеше происходит следующим образом: при получении ответа от вышестоящего сервера nginx сохраняет его во временный файл вне структуры каталогов кеша. После завершения обработки запроса nginx переименовывает временный файл и перемещает его в каталог с кешем. Если каталог для хранения временных файлов располагается в другой файловой системе, то он будет скопирован. Поэтому рекомендуется размещать каталог для временных файлов и кеш в одной файловой системе. Также вполне безопасно удалять файлы из каталога кеша в случае, когда явно видно, что они должны быть очищены. Также существуют сторонние расширения для nginx, которые позволяют дистанционно контролировать содержимое кеша, и в будущем этот функционал планируется интегрировать в основной дистрибутив.

14.3 Настройки nginx

Вдохновение при создании системы конфигурирования nginx Игорь Сысоев черпал в Apache. Его главная идея состояла в том, что масштабируемая система конфигурации имеет важнейшее значение для web-сервера. Основная проблема масштабируемости проявилась при поддержании боль-

ших и сложных конфигураций с большим количеством виртуальных серверов, каталогов, корневых директорий web-сайтов (*locations*) и наборов данных. В сравнительно больших инсталляциях эта проблема может стать кошмаром, если всё сразу не сделано должным образом, как на уровне приложения, так и на уровне системного инженера.

В результате, в nginx была разработана система конфигурации, упрощающая повседневные операции и обеспечивающая простые средства для дальнейшего расширения настроек web-сервера.

Конфигурационные файлы nginx хранятся в файлах, обычно расположенных в каталоге `/usr/local/etc/nginx` или `/etc/nginx`. Основной конфигурационный файл обычно называется `nginx.conf`. Для достижения лаконичности часть настроек может быть помещена в отдельные файлы, которые будут автоматически объединены с основным на этапе загрузки. Тем не менее, следует отметить, что в настоящее время nginx не поддерживает распределённые конфигурации в стиле Apache (например, файлы `.htaccess`). Все настройки, определяющие поведение nginx, должны храниться в упорядоченном наборе конфигурационных файлов.

Перед загрузкой управляющий процесс (*master*) проверяет конфигурационные файлы на доступность для чтения и корректность. Перед выделением из управляющего потока каждому исполнителю предоставляется доступ к уже скопированному в один конфигурационному файлу, доступному им только для чтения. Данные о настройках автоматически становятся доступными благодаря использованию обычных механизмов управления виртуальной памятью.

Файл настроек nginx может содержать несколько типов блоков директив: основные настройки (*main*), настройки для http (*http*), настройки задания виртуальных серверов (*server*), настройки для указания вышестоящих серверов (*upstream*), настройки определения местоположения web-сайта (*location*) и настройки для проксирования электронной почты (*mail*). Группы настроек никогда не пересекаются. Например, не допускается помещать блок директивы *location* в блок основных настроек *main*. Кроме того, для исключения двусмысленностей отсутствуют такие настройки, как "глобальный web-сервер". Настройки nginx подразумевают чистоту и логичность, позволяя пользователям поддерживать сложные конфигурационные файлы с тысячами директив. В частной беседе Сысоев сказал: "Locations, directories и другие блоки настроек в общей конфигурации, как это сделано в Apache, мне никогда не нравились и это является причиной, по которой они никогда не появятся в nginx".

Синтаксис, форматирование и определения основываются на так называемом Си-стиле (C-style). Такой приём построения конфигурационных файлов уже используется некоторыми проектами с открытым исходным кодом и коммерческими приложениями. По задумке, Си-стиль хорошо подходит для определения вложенных описаний, являясь логичным и простым для создания, чтения и сопровождения, за что его и полюбили многие инженеры. Также Си-стиль конфигурации nginx легко поддаётся автоматизации.

Хотя некоторые директивы nginx напоминают конфигурацию Apache, настройка nginx - это совсем другой опыт. Например, nginx поддерживает переопределение правил, что в случае с настройкой Apache потребовало бы ручной правки конфигурационных файлов. Реализации способов перезаписи также отличаются.

В целом, настройки nginx поддерживают несколько оригинальных механизмов, которые могут быть очень полезны при конфигурировании web-сервера с минимальными требованиями. Имеет смысл упомянуть именованные переменные (*variables*) и директиву `try_files`, которые в некоторой степени уникальны. Именованные переменные были реализованы для обеспечения ещё более мощного механизма управления web-сервером на этапе загрузки конфигурации. Именованные переменные уже пре-компилированы и оптимизированы для быстрого разрешения в значения. Разрешение переменной в значение выполняется по требованию, то есть значение, как правило, рассчитывается один раз и остаётся неизменным в кеше всё время жизни конкретного запроса. Имено-

ванные переменные могут использоваться с различными конфигурационными директивами, что обеспечивает дополнительную гибкость при описании реакции на запросы по условию.

Директива `try_files` изначально была предназначена для постепенной замены условного оператора `if`, так как более правильно, быстро и эффективно "попробовать-сравнить" чем поддерживать карты соответствия между URI и данными. И в целом, директива `try_files` работает хорошо, может быть весьма эффективной и полезной. Можно порекомендовать читателю использовать директиву `try_files` там, где это применимо.

14.4. Внутреннее устройство nginx

Как уже упоминалось ранее, nginx состоит из ядра и некоторого количества модулей. Ядро nginx отвечает за базовый функционал web-сервера и функционал обратного проксирования web и электронной почты, что позволяет предоставлять доступ к реализованным в ядре сетевым протоколам, создавать необходимые среды исполнения и обеспечивать "бесшовное" взаимодействие между модулями. Тем не менее, большинство функций, специфичных для протоколов и приложений, реализуется модулями, а не ядром.

Внутри себя nginx обрабатывает соединения с помощью каналов (pipeline), цепочек команд (chain) или модулей. Другими словами, для каждой операции находится модуль, который и выполняет соответствующую работу (например, сжатие, преобразование данных, выполнение серверных сценариев, взаимодействие с вышестоящими серверами приложений с применением FastCGI или uwsgi протоколов, взаимодействие с memcached).

Есть два модуля, размещённые где-то между ядром и модулями с реальным функционалом. Это модули `http` и `mail`. Эти модули обеспечивают дополнительный уровень абстракции между ядром и низкоуровневыми компонентами. В них реализована обработка последовательностей событий, связанная с протоколами прикладного уровня HTTP, SMTP или IMAP. Вместе с ядром эти модули обеспечивают поддержание правильного порядка вызовов соответствующих функциональных модулей. В то время как протокол HTTP реализован в виде части модуля `http`, уже есть планы по реализации его в виде отдельных функциональных модулей в связи с необходимостью поддержки других протоколов, например, SPDY (смотрите "[SPDY: An experimental protocol for a faster web](#)").

Функциональные модули можно разделить на следующие типы: модули обработки событий, модули обработки фазы, выходные фильтры, модули обработки именованных переменных, модули реализации протоколов, модули взаимодействия с вышестоящими серверами и балансировщики нагрузки. Большинство из этих модулей дополняют HTTP-функциональность nginx, хотя модули обработки событий и реализации протоколов также используются и в модуле `mail`. Модули обработки событий реализуют зависимые от операционной системы механизмы извещения о событиях, например, `kqueue` или `epoll`. Модуль обработки событий, используемый в nginx, зависит от операционной системы, на которой он запущен, и настроек. Модули поддержки протоколов позволяют nginx взаимодействовать с использованием HTTP, TLS/SSL, SMTP, POP3 и IMAP.

Типовой цикл обработки HTTP-запроса выглядит следующим образом:

1. Клиент посылает HTTP-запрос
2. Ядро nginx выбирает соответствующий обработчик фазы на основе сопоставления содержимого запроса и настроенных корневых каталогов web-сайтов (`location`)
3. Если `location` настроен в качестве балансировщика нагрузки, то nginx выбирает вышестоящий сервер для проксирования
4. Обработчик фазы выполняет свою работу и каждый выходной буфер от него подаётся на вход первого фильтра
5. Обработчик фазы первого фильтра подаёт данные на второй фильтр

6. Обработчик фазы второго фильтра подёт данные на третий и т.д.
7. Подготовленный ответ отсылается клиенту

Порядок вызова модулей в nginx чрезвычайно настраиваемый. Он осуществляется с помощью ряда функций обратного вызова с указателями на исполняемые функции. Однако, недостатком такого подхода является высокий входной порог для программистов, которые хотели бы писать свои собственные модули, так как им потребуется разобраться в этом и точно указывать как и когда модуль должен работать. Для облегчения этой нагрузки постоянно улучшаются API и документация для разработчиков.

Несколько примеров мест, где могут быть подключены модули:

- перед считыванием и применением конфигурационного файла
- для каждой директивы `location` и `server` везде, где она встретится
- момент применения основной конфигурации
- момент инициализации сервера (например, хоста или порта)
- момент объединения конфигурационных файлов с основной конфигурацией
- момент инициализации секции `location` или её объединения с конфигурацией "родительского" сервера
- момент запуска и останова управляющего процесса
- момент запуска или завершения исполнителя
- момент обработки запроса
- момент фильтрации заголовка или содержимого ответа
- момент выбора вышестоящего сервера, инициирования запроса к нему и повторного запроса
- момент обработки ответа от вышестоящего сервера
- момент завершения взаимодействия с вышестоящим сервером

Последовательность действий цикла генерации ответа внутри исполнителя выглядит следующим образом:

1. Запуск цикла `ngx_worker_process_cycle()`
2. Обработка событий с использованием зависимых от операционной системы механизмов (таких, как `epoll` или `kqueue`)
3. Приём событий и управление выполнением соответствующих им действий
4. Обработка или проксирование заголовка или содержимого запроса
5. Генерация содержимого ответа (заголовок и данные) и передача его клиенту
6. Завершение обработки запроса
7. Сброс таймеров и событий

Шаги 5 и 6 выполнения цикла обработки запроса обеспечивают поэтапную генерацию ответа и передачу клиенту.

Более детальное описание процесса обработки HTTP-запроса может выглядеть следующим образом:

1. Инициализация процесса обработки
2. Обработка заголовка
3. Обработка данных запроса
4. Вызов соответствующего обработчика
5. Переход от фазы к фазе при обработке

Зачем нужны фазы. При обработке HTTP-запроса nginx проводит его через серию фаз обработки. С каждой фазой могут быть ассоциированы и вызваны обработчики. Таким образом, обработчики,

ассоциированные с фазами, выполняют обработку запроса и формирование соответствующего ответа. Соответствие фаз и обработчиков задаётся в конфигурационном файле.

Обработчики фазы обычно выполняют четыре задачи: считывают расположение корневого каталога web-сайта (`location`), генерируют соответствующие ответы на запросы, выполняют отправку заголовков и данных ответа. Обработчик имеет всего один аргумент: структура с описанием запроса. В структуре описания запроса определяется много полезной информации о запросе клиента: метод запроса, URI и заголовок.

Во время чтения заголовка HTTP-запроса nginx выполняет поиск соответствующего виртуального сервера согласно конфигурации. Если виртуальный сервер найден, то запрос проходит шесть фаз:

1. преобразование URI на уровне сервера
2. поиск конфигурации в которой будет обрабатываться запрос
3. преобразование URI на уровне `location` (что может привести к запросу на возврат к предыдущей фазе)
4. проверка доступа
5. обработка директив `try_files`
6. журналирование (запись лога)

Для создания ответа на запрос nginx попытается передать его подходящему обработчику генерирования ответа. В зависимости от настроек `location` nginx может попробовать безусловные обработчики `perl`, `proxy_pass`, `flv`, `mp4` и т.д. Если запрос не соответствует ни одному из вышеперечисленных обработчиков, то будут последовательно перебраны следующие модули в указанном порядке: `random_index`, `index`, `autoindex`, `gzip_static`, `static`.

Более подробное описание модулей `index` приведено в документации nginx. Эти модули отвечают за обработку запросов на адреса, завершающиеся слешем. Если специализированные модули не подходят (например, `mp4` или `autoindex`), то запрашиваемые данные считаются файлом или каталогом (то есть, статическими данными) и обрабатываются обработчиком `static`. Для каталога URI будет автоматически преобразован в адрес, завершающийся слешем, и затем выполнится перенаправление HTTP-запроса.

Данные, сгенерированные обработчиком, передаются на фильтры. Задание ассоциаций для фильтров также выполняется на уровне `location` с возможностью настройки сразу нескольких фильтров. Фильтры выполняют задачу дополнительной обработки сгенерированного обработчиками потока. Порядок запуска фильтра определяется на этапе компиляции. Все фильтры, как поставляемые вместе с nginx, так и сторонние, могут быть настроены на этапе сборки. В текущей реализации nginx фильтры могут быть применены только к выходным данным. Механизма применения фильтров к входным данным в настоящее время не существует. Применение фильтров к входному потоку появится в будущих версиях nginx.

Фильтры строятся согласно определённому шаблону проектирования. Фильтр вызывается, начинает работу, вызывает следующий фильтр и так до тех пор, пока не будет вызван последний фильтр в этой цепочке. После этого nginx завершает подготовку ответа. Следующему фильтру не обязательно ждать, пока предыдущий завершит работу. Следующий фильтр может начинать работу сразу, как только появится порция обработанных данных предыдущим фильтром (по аналогии с каналами (pipeline) Unix). В свою очередь, генерируемый на выходе последнего фильтра ответ может быть передан клиенту до получения полного ответа от вышестоящего сервера.

Фильтры разделяются на фильтры заголовков и данных. nginx при подготовке ответа на запрос подаёт заголовки и данные на соответствующие фильтры раздельно.

Фильтрация заголовка состоит из трёх основных шагов:

1. Принятие решения о реакции на запрос
2. Обработка ответа
3. Вызов следующего фильтра

Фильтры данных преобразуют содержимое ответа. Могут быть приведены следующие примеры фильтров:

- серверные включения
- фильтрация XSLT
- фильтрация изображений (например, изменение размера "на лету")
- конвертирование кодировки
- сжатие gzip
- передача данных с использованием механизма `Chunked transfer encoding`

После завершения работы цепочки фильтров ответ передаётся на отправку. Одновременно с отправкой могут работать два специальных фильтра `copy` и `postpone`. Фильтр `copy` отвечает за заполнение буферов в оперативной памяти соответствующим ответом, который может храниться во временном каталоге прокси. Фильтр `postpone` используется для выполнения подзапросов.

Подзапросы являются очень важным механизмом обработки по типу "запрос-ответ". Подзапросы также являются одним из самых мощных механизмов nginx. В результате применения подзапросов nginx может вернуть в качестве результата другой URL, а не тот, что запрашивал клиент. В некоторых библиотеках разработки (frameworks) это называется внутренним перенаправлением. Однако, nginx идёт ещё дальше - помимо выполнения нескольких подзапросов с несколькими фильтрами и объединением всего этого в один ответ, подзапросы могут быть вложенными и выстроеными в иерархическую структуру. Подзапросы могут выполнять свои под-подзапросы, которые в свою очередь могут инициировать под-под-подзапросы. Подзапросы можно связать с файлами на диске, другими обработчиками или вышестоящими серверами. Подзапросы - это очень полезный механизм, позволяющий дополнять ответ, выполняя под-запросы на основе данных подготовленного ранее ответа. Например, модуль SSI (серверные включения) использует фильтр для анализа содержимого возвращаемого документа, а затем подменяет директивы `include` к конкретным URL. Или может быть приведён ещё такой пример: использование фильтра для извлечения документа по указанной ссылке URL, его обработка, сохранение и возвращение ссылки URL на уже обработанный документ.

Перенаправление запросов (`upstream`) и проксирование также заслуживают упоминания. Перенаправление можно описать как совокупность обработчика данных запроса и обратного прокси (`proxy_pass`). Upstream-модули предназначены в основном для отправки запросов на вышестоящий сервер (или встроенную подсистему обработки - backend) и получения ответов от него. Здесь вызов фильтров не предусмотрен. Что точно делает upstream-модуль, так это запускает функции обратного вызова, когда вышестоящий сервер готов для записи или чтения. В nginx реализованы следующие функции обратного вызова:

1. создание буфера запросов (или цепочек из них), который будет направлен в вышестоящий сервер
2. повторная инициализация или сброс соединения с вышестоящим сервером (что происходит непосредственно перед повторным запросом)
3. обработка первых битов ответа вышестоящего сервера и сохранение указателей на полученные от него данные
4. прерывание запросов (что происходит при преждевременном завершении работы клиента)
5. завершение обработки запроса при получении всех данных от вышестоящего сервера
6. удаление части ответа (например, завершающей части пакета - trailer)

Модули балансировки нагрузки дополняют обработчик `proxy_pass` и предоставляют возможность выбора вышестоящего сервера в случае доступности нескольких. Балансировщик нагрузки включается соответствующей директивой в конфигурационном файле, обеспечивает дополнительный функционал при инициализации вышестоящего сервера (разрешение вышестоящего сервера по DNS-имени и т.д.), инициализирует структуры для описания соединения, принимает решение о выборе вышестоящего сервера для перенаправления запроса и обновляет статистику. В настоящее время nginx поддерживает два способа балансировки нагрузки: циклический (round-robin) и IP-хеш (IP-hash).

Upstream-модули и модули балансировки включают в себя механизм обнаружения сбоев вышестоящих серверов и автоматического перенаправления запросов на оставшиеся серверы. В будущих версиях планируется много работы для дальнейшего развития этого функционала. В частности, планируется значительно улучшить механизмы проверки состояния вышестоящих серверов и распределение нагрузки между ними в соответствии с полученной оценкой при проверке.

Также есть несколько других интересных модулей, обеспечивающих применение дополнительного набора именованных переменных (`variables`) в конфигурационном файле. В то время, как в различных модулях создаются и обновляются наборы именованных переменных, есть два модуля, полностью посвящённых именованным переменным `geo` и `map`. Модуль `geo` используется для облегчения отслеживания клиентов по их IP-адресам. Этот модуль может создавать различные именованные переменные в зависимости от IP-адреса клиента. Модуль `map` позволяет создавать именованные переменные на основе других именованных переменных, предоставляя, по сути, гибкий механизм отображения имён хостов и динамических переменных. Этот тип модулей может быть назван - "обработчик именованных переменных".

В некоторой степени Apache повлиял на создание в nginx механизма выделения памяти внутри исполнителя. В общем виде описание механизма управления памятью в nginx имеет следующий порядок: для каждого соединения динамически выделяются необходимые буферы; для них устанавливается соответствие с соединением; буферы используются для хранения, обработки заголовка и данных запроса, отправки ответа; после завершения соединения память освобождается. Важно отметить, что nginx пытается избежать копирования данных в оперативной памяти и максимально возможно использует передачу по указателю на значение вместо вызова функции `memcpy`.

Если немного подробнее, то после генерирования ответа с помощью модуля полученный результат помещается в буфер памяти, который добавляется в цепочку буферов, ассоциированных с соединением. Последующие обработки работают с этой же цепочкой буферов. Цепочки буферов в nginx устроены довольно сложно, так как есть несколько сценариев их работы, зависящих от типа модуля. Например, может оказаться довольно сложным управление буферами при реализации модуля фильтрации данных запроса. Этот модуль должен работать только с одним буфером (звеном цепочки) и в то же время он должен решить, следует ли перезаписать входной буфер, заменить его на новый или вставить в цепочку другой буфер перед или после него. Может быть ещё сложнее: иногда модуль получает несколько буферов и ему приходится обрабатывать неполную цепочку буферов. Тем не менее, nginx в настоящее время для работы с цепочками буферов предоставляет только низкоуровневый API. Поэтому для реализации сторонних модулей разработчик должен очень хорошо ориентироваться в этой мистической части nginx.

Отметим, что всё время жизни соединения ему соответствует ряд буферов памяти. То есть, длительными соединениями блокируется некоторая часть оперативной памяти. В то же время, на поддержание простоявшего соединения nginx тратит всего 550 байт оперативной памяти. В качестве оптимизации в будущих версиях nginx может быть реализовано повторное и совместное использование буферов длительных соединений.

Задача управления оперативной памятью в nginx решается распределителем пула (pool allocator). Области разделяемой памяти используются для организации мьютексов, кеша метаданных, кеша

SSL-сессий и хранения информации об управлении пропускной способностью (лимиты). В nginx для управления разделяемой памятью используется slab-распределитель. Для потокобезопасного использования разделяемой памяти используются механизмы блокировки доступа (мьютексы и семафоры). Также при организации сложных структур данных в nginx используются красно-чёрные деревья (red-black tree). Красно-чёрные деревья используются для хранения кеша метаданных в разделяемой памяти, отслеживания нерегулярных значений location и некоторых других задач.

К сожалению, всё вышеперечисленное не было документировано в последовательной и простой манере, что делает разработку сторонних модулей для nginx довольно сложной. Тем не менее, некоторые хорошие материалы существуют. Например, материалы, изданные Эваном Миллером (Evan Miller). Такие материалы требуют огромной работы по обратному инжинирингу и изучению работы модулей, что для многих сравнимо с чёрной магией.

Несмотря на определённые трудности, связанные с разработкой сторонних модулей, в сообществе пользователей nginx существует большое количество полезных модулей. Например, встроенный интерпретатор с языка Lua, дополнительные модули балансировки нагрузки, модуль полной поддержки WebDAV, модуль расширенного управления кешем и другие интересные модули, что авторы этой главы поощряют и будут поддерживать в будущем.

14.5. Выводы

Когда Игорь Сысоев начал разработку nginx, большинство программного обеспечения, реализующего Интернет, уже существовало. Архитектура этого программного обеспечения, как правило, соответствовала устаревшим подходам построения серверов, сетевого оборудования, операционных систем и старой архитектуре Интернета в целом. Тем не менее, это не помешало Игорю считать, что он мог бы улучшить положение вещей в области web-серверов. Таким образом, первый вывод может показаться очевидным, но он следующий: всегда есть место для совершенствования.

Постоянно держа в голове идею улучшения web-программного обеспечения, Игорь потратил много времени на разработку исходной структуры кода и изучение способов оптимизации кода для различных операционных систем. Десять лет спустя он разрабатывает прототип версии 2.0 с учётом лет активного развития версии 1.0. Понятно, что прототип новой архитектуры и исходная структура кода имеют жизненно важное значение для будущего программного продукта.

Ещё один момент, который стоит отметить, это необходимость в направленности развития. Windows-версия nginx, вероятно, хороший пример того, как стоит избегать рассосредоточения усилий на то, что не находится в компетенции разработчика или не является основной целью приложения. Это в равной степени относится и к переписыванию ядра, что было сделано в течении нескольких попыток добавления новых возможностей nginx для улучшения обратной совместимости со старыми версиями.

Последнее, но не менее важное, что стоит отметить, это то, что несмотря на небольшое сообщество разработчиков nginx, сторонние модули и расширения всегда были очень важной составляющей его популярности. Работа, выполненная Evan Miller, Piotr Sikora, Valery Kholodkov, Zhang Yichun (agentzh) и другими талантливыми разработчиками высоко ценится сообществом nginx и его непосредственными разработчиками.

15. Open MPI

15.1. Введение

Open MPI [GFB +04] - это программная реализация стандарта интерфейса передачи сообщений (MPI) с открытым исходным кодом. Для того, чтобы можно было рассматривать архитектуру и внутреннюю организацию Open MPI, нужно немного обсудить стандарт MPI.

Интерфейс передачи сообщений (MPI)

Стандарт MPI создан и поддерживается форумом [MPI Forum](#) - открытой группой, состоящей из экспертов по параллельным вычислениям, причем как из производственных, так и из научных кругов. В стандарте MPI определяется интерфейс API, который используется для переносимого высокопроизводительного межпроцессного взаимодействия (IPC) определенного типа: *передачи сообщений*. В частности, в документе MPI описывается надежная передача дискретных сообщений между процессами MPI. Хотя определение «процесс MPI» подлежит некоторой интерпретации для конкретной платформы, он, как правило, соответствует концепции процесса операционной системы (например, процесса POSIX). Интерфейс MPI специально предназначен реализации в среднем слое, что означает, что приложения, находящиеся выше, вызывают функции MPI для передачи сообщений.

MPI определяет высокоуровневый интерфейс API, что означает, что в нем абстрагируются от всех лежащих ниже транспортных механизмов, используемых при передаче сообщений между процессами. Идея состоит в том, чтобы процесс X, отправляющий сообщение, мог, по сути, сказать следующее: "берем этот массив из 1073 значений двойной точности и отправляем его процессу Y". Соответствующий процесс Y, получающий сообщение, по сути, мог сказать: "получаем массив из 1073 значений двойной точности от процесса X". Происходит чудо и массив из 1073 значений двойной точности поступает в ожидающий его буфер в процессе Y.

Обратите внимание на то, чего нет в этом обмене: нет такого понятия, как создание подключения, нет потока байтов, который нужно интерпретировать и нет сетевых адресов, используемых при обмене. Интерфейс MPI, абстрагируясь от всего этого, не только скрывает всю эту сложность от приложения, находящегося на более высоком уровне, но и делает приложение переносимым в другие среды и на другие транспортные уровни, осуществляющие передачу сообщений. В частности, правильное приложение MPI является совместимым по исходному коду в широком спектре платформ и типов сетей.

В интерфейсе MPI определяется не только соединение типа «точка-точка» (например, отправка и получение сообщения), в нем также определяются другие шаблоны соединения, например, *коллективные (collective)* соединения. Коллективными операциями являются такие, при которых в одном действии коммуникации участвуют несколько процессов. Например, надежное широковещательная передача данных (broadcast), когда в начале операции сообщение есть у одного процесса, а в конце операции это сообщение есть у всех процессов в группе. В MPI также определены другие концепции и шаблоны коммуникаций, которые здесь не описываются. На момент написания статьи последней версией стандарта MPI был стандарт MPI-2.2 [For09]. Также были опубликованы черновые версии нового стандарта MPI-3; он должен быть опубликован уже в конце 2012 года. *Прим. пер.: эта версия была опубликована 21 сентября 2012 года.*

Использование MPI

Есть много реализаций стандарта MPI, в которых поддерживается широкий спектр различных платформ, операционных систем и типов сетей. В некоторых реализациях открытый исходный код, некоторые из них - закрытый. Open MPI, как следует из названия, является одной из реализаций с открытым исходным кодом. К числу типичных транспортных сетей MPI относятся следующие (но ими не ограничиваются): различные протоколы поверх Ethernet (например, TCP, iWARP, UDP, сами фреймы Ethernet и т.д.), совместно используемая память и InfiniBand.

Реализации MPI обычно используются в так называемых средах «высокопроизводительных вычислений» (HPC). Интерфейс MPI, в сущности, предоставляет соединения типа IPC для программ моделирования, вычислительных алгоритмов и других приложений типа «больших числовых молотилок». Для входных данных, с которыми работают эти приложения, обычно требуется выполнить слишком большой объем вычислений с тем, чтобы можно было ограничиться только одним сервером; задания MPI распределены по десяткам, сотням, а то и тысячам серверов, причем для того, чтобы решить одну вычислительную задачу, все они работают сообща.

Это означает, что приложения, использующие MPI, являются по своей сути параллельными и требуют высокой вычислительной мощности. Нет ничего необычного в том, что все ядра процессора при выполнении задания MPI работают на 100%. Для ясности — задания MPI обычно выполняются в специализированных средах, где процессы MPI являются единственным приложением, запущенным на машине (конечно, в дополнение к минимальным функциональным возможностям операционной системы).

Таким образом, реализации MPI, как правило, ориентированы на обеспечение чрезвычайно высокой производительности, измеряемой такими показателями, как:

- Чрезвычайно низкими задержками при передаче коротких сообщений. В качестве примера, 1-байтовое сообщение может быть отправлено из процесса Linux пользовательского уровня через коммутатор InfiniBand и получено в целевом процессе Linux пользовательского уровня на другом сервере за время, чуть более 1 микросекунды (т.е., 0,000001 секунды).
- Чрезвычайно высокая скорость отправки сообщений в сеть при передаче коротких сообщений. Некоторые поставщики имеют реализации MPI (в паре со специальным оборудованием), которые могут отправлять в сеть до 28 млн. сообщений в секунду.
- Быстрый разгон (в зависимости от размера сообщения) до максимальной пропускной способности, поддерживаемой базовым транспортным механизмом.
- Незначительное потребление ресурсов. Все ресурсы, используемые MPI (например, память, кэш и пропускная способность шины) не могут использоваться в приложении. Поэтому реализации MPI пытаются соблюсти баланс незначительного потребления ресурсов, одновременно обеспечивая высокую производительность.

Open MPI

Первая версия стандарта MPI - MPI-1.0 была опубликована в 1994 г. [Mes93]. Версия MPI-2.0, являющаяся набором дополнений поверх стандарта MPI-1, была завершена в 1996 г. [GGHL +96].

В первые десять лет после публикации стандарта MPI-1 количество различных реализаций MPI увеличилось. Некоторые из них представлялись поставщиками для своих собственных средств межсетевых соединений. Другие реализации возникли в среде исследовательских и академических сообществ. Такие реализации были по качеству типично «исследовательскими», что означает, что их целью было изучение различных концепций высокопроизводительных сетей и представление доказательств концепций правильности их работы. Тем не менее, некоторые из них были достаточно высокого качества, так что они завоевали популярность и привлекли некоторое количество пользователей.

Open MPI представляет собой объединение четырех исследовательских/академических реализаций MPI с открытым исходным кодом: LAM/MPI, LA/MPI (Лос-Аламосский вариант MPI) и FT-MPI (отказоустойчивый вариант MPI). Вскоре после создания группы Open MPI к ней присоединилась команда проекта PACX-MPI.

Когда к нам пришло коллективное осознание того, что, кроме незначительных различий в оптимизации и возможностях, наши варианты кода программ кода были очень похожи, представители этих четырех команд разработчиков решились на сотрудничество. Каждый из четырех вариантов

кода имел свои сильные и слабые стороны, но в целом, они делали более или менее одно и то же. Так зачем конкурировать? Почему бы не объединить наши ресурсы, работать вместе, и сделать еще лучшую реализацию MPI?

После долгих обсуждений было принято решение отказаться от наших четырех уже существующих вариантов кода и взять из предыдущих проектов только лучшие идеи. Это решение было обусловлено главным образом следующими соображениями:

- Хотя во всех четырех вариантах кода многие из основных алгоритмов и методик были аналогичными, каждый из них имел радикально отличающуюся архитектуру реализации, и объединять их было бы невероятно трудно (если это и было возможным).
- Каждый из четырех вариантов также имели свои собственные (существенные) сильные и (существенные) слабые стороны. В частности, в каждом из четырех вариантов были функции и архитектурные решения, разработку которых желательно было бы продолжить. Кроме того, в каждом из четырех вариантов был плохо оптимизированный и плохо сделанный код, который желательно было выбросить.
- Представители четырех групп разработчиков раньше вместе непосредственно не работали. Начало разработки с совершенно нового варианта кода (а не улучшение одного из существующих вариантов) ставило всех разработчиков в равные условия.

Таким образом, проявился проект Open MPI. Первый раз он был помещен в Subversion 22 ноября 2003 года.

15.2. Архитектура

По целому ряду причин (в основном, связанных либо с производительностью, либо с переносимостью) единственными двумя возможностями языка первичной реализации были C и C++. Язык C++ в конце концов был отвергнут, поскольку различные компиляторы C++, как правило, размещали структуры/классы в памяти в соответствии с различными алгоритмами оптимизации, что при работе с сетью приводило к различным реализациям. Поэтому в качестве основного языка реализации был выбран язык C, что оказало влияние на несколько архитектурных проектных решений.

Когда проект Open MPI был запущен, мы знали, что он должен представлять собой сложный код большого объема:

- В 2003 году в MPI-2.0, текущей версии стандарта MPI на тот момент, было определено более 300 функций API.
- Каждый из четырех предыдущих проектов был большим сам по себе. Например, LAM/MPI имел более 1900 файлов исходного кода, насчитывающего более 300 000 строк кода (считая строки комментариев и пробелов).
- Мы хотели, чтобы Open MPI поддерживал больше функций, сред и сетей, чем все четыре предыдущих проекта, взятые вместе.

Поэтому на разработку архитектуры мы потратили много времени, причем сосредоточивались на следующих трех аспектах:

1. Аналогичные функции группировались вместе в виде отдельных слоев абстракции.
2. Для выбора различных реализаций одного и того же варианта поведения системы использовались загружаемые плагины и параметры времени выполнения.
3. Не допускалось, чтобы абстракция влияла на способы исполнения.

Архитектура слоев абстракции

В Open MPI есть три основных слоя абстракции, которые показаны на рис.15.1:

- *Open, Portable Access Layer (OPAL)*: Слой OPAL является нижним слоем абстракции проекта Open MPI. Его абстракции сфокусированы на отдельных процессах (а не параллельном выполнении заданий). Он предоставляет утилиты и связующий код, например, связные списки общего назначения, обработку строк, управление отладкой и другие рутинные, но необходимые функции.

В слое OPAL также реализуется ядро переносимости Open MPI между различными операционными системами, например, доступ к интерфейсам IP, совместное использование память на одном и том же сервере, согласование процессора и памяти, высокоточные таймеры и т.д.

- *Open MPI Run-Time Environment (ORTE) (произносится как «ор-тей»)*: Реализация MPI должна предоставить не только интерфейс API, необходимый для передачи сообщений, но и сопутствующую систему времени выполнения для запуска, отслеживания и уничтожения параллельно выполняемых заданий. В случае Open MPI параллельно выполняемое задание состоит из одного или нескольких процессов, которые могут исполняться на нескольких экземплярах операционной системы и могут быть взаимосвязаны друг с другом так, чтобы они действовали как нечто единое.

В простых средах со слабой поддержкой или без поддержки распределенных вычислений слой ORTE использует команды `rsh` или `ssh` для запуска отдельных процессов в параллельно выполняемых заданиях. В более продвинутых HPC-средах обычно есть планировщики и менеджеры ресурсов, применяемые для справедливого распределения вычислительных ресурсов между многими пользователями. В таких средах обычно предоставляются специализированные интерфейсы, предназначенные для запуска и регулирования процессов на вычислительных серверах. В слое ORTE поддерживается широкий спектр таких управляемых сред, например (но не только): Torque/PBS Pro, SLURM, Oracle Grid Engine и LSF.

- *Open MPI (OMPI)*: Слой MPI является самым высоким уровнем абстракции, и является единственным слоем, который виден приложениям. В этом слое реализован интерфейс API для MPI, т.к. в нем заключена семантика передачи сообщений, определяемая стандартом MPI.

Поскольку переносимость является основным требованием, в слое MPI поддерживается широкий спектр различных типов сетей и лежащих в их основе протоколов. Некоторые сети похожи по своим основным характеристикам и абстракциям, а некоторые - нет.



Рис.15.1: Представление архитектуры абстрактных слоев проекта Open MPI в виде трех основных слоев: OPAL, ORTE и OMPI

Хотя каждая абстракция представляет собой слой, расположенный поверх слоя, лежащего ниже, по причинам, связанным с производительностью, слои ORTE и OMPI могут, когда необходимо, обходить нижележащие слои абстракции и непосредственно взаимодействовать с операционной системой и/или аппаратным обеспечением (так, как показано на рис.15.1). Например, для достижения максимальной производительности сетей в слое OMPI используются методы обхода ОС при взаимодействии с определенными типами аппаратных интерфейсов NIC.

Каждый слой собран в виде отдельной библиотеки. Библиотека ORTE зависит от библиотеки OPAL; библиотеки OMPI зависят от библиотеки ORTE. Разделение слоев на свои собственные библиотеки стало прекрасным инструментом для предотвращения нарушений абстракции. В частности, приложения не смогут быть скомпонованы, если некоторый слой пытается неправильно использовать символ, находящийся на более высоком уровне. На протяжении многих лет такой механизм абстракции защищал многих разработчиков от случайного нарушения границ между этими тремя слоями.

Архитектура плагинов

Хотя первоначально члены сообщества Open MPI стремились к одной и той же основной цели (создать переносимую высокопроизводительную реализацию стандарта MPI), наши организационные возможности, мнения и декларации, да и все тому подобное, были абсолютно различными. Поэтому мы потратили достаточно много времени на разработку архитектуры, которая позволила бы нам оставаться разными даже в случае совместного использования одного и того же базового кода.

Естественным выбором стали компоненты, загружаемые во время исполнения (т.е. динамически разделяемые объекты или «DSO», или «плагины»). Компоненты соответствовали общему интер-

файсу API, но они имели незначительные ограничения на реализацию этого API. А именно: одно и то же поведение интерфейса можно было реализовывать несколькими способами. Пользователь мог на этапе исполнения выбрать, какой плагин (плагины) он будет использовать. Это даже позволило третьим лицам самостоятельно разрабатывать и распространять свои собственные плагины Open MPI, которые не входят в состав базового пакета Open MPI. Возможность произвольного расширения является вполне либеральной политикой, причем как непосредственно среди разработчиков Open MPI, так и в гораздо большем по размеру сообществе Open MPI.

Такая гибкость времени выполнения является ключевым компонентом философии проекта Open MPI и она глубоко интегрирована во всей его архитектуре. Показательный пример: серия Open MPI v1.5 включает в себя 155 плагинов. Просто перечислим лишь несколько примеров: есть плагины для различных реализаций `memcopy()`, плагины для дистанционного запуска процессов на других серверах, и плагины для взаимодействия в различных типах базовых сетей.

Одно из основных преимуществ использования плагинов состоит в том, что несколько групп разработчиков могут свободно экспериментировать с альтернативными реализациями, не затрагивая основной проект Open MPI. Это была очень важно особенно в первое время работы над проектом Open MPI. Иногда разработчики не знают, как правильно что-то реализовать, а иногда просто не соглашаются друг с другом. В обоих случаях, каждая из сторон будет реализовывать свое собственное решения в виде компонента, позволив остальной части сообщества разработчиков легко сравнивать и сопоставлять результаты. Конечно, сравнение кода может быть выполнено без использования компонентов, но концепция компонентов позволяет гарантировать, что все реализации будут находиться в условиях одного и того же внешнего API, и, следовательно, будет обеспечена одна и та же необходимая семантика.

Прямыми результатом такой гибкости является то, что она обеспечивает, что компонентная концепция используется в полной мере во всех трех слоях проекта Open MPI; в каждом слое есть много различных типов компонентов. Каждый тип компонента представлен в виде *фреймворка*. Компонент принадлежит ровно одному фреймворку, а фреймворк поддерживает ровно один вид компонента. На рис.15.2 приведена общая компоновка архитектуры проекта Open MPI; на ней показаны несколько фреймворков Open MPI и некоторые из имеющихся компонентов. Остальные фреймворки и компоненты Open MPI подключены к проекту аналогичным образом. Набор слоев проекта Open MPI, его фреймворки и компоненты называются модульной архитектурой компонентов - Modular Component Architecture (MCA).

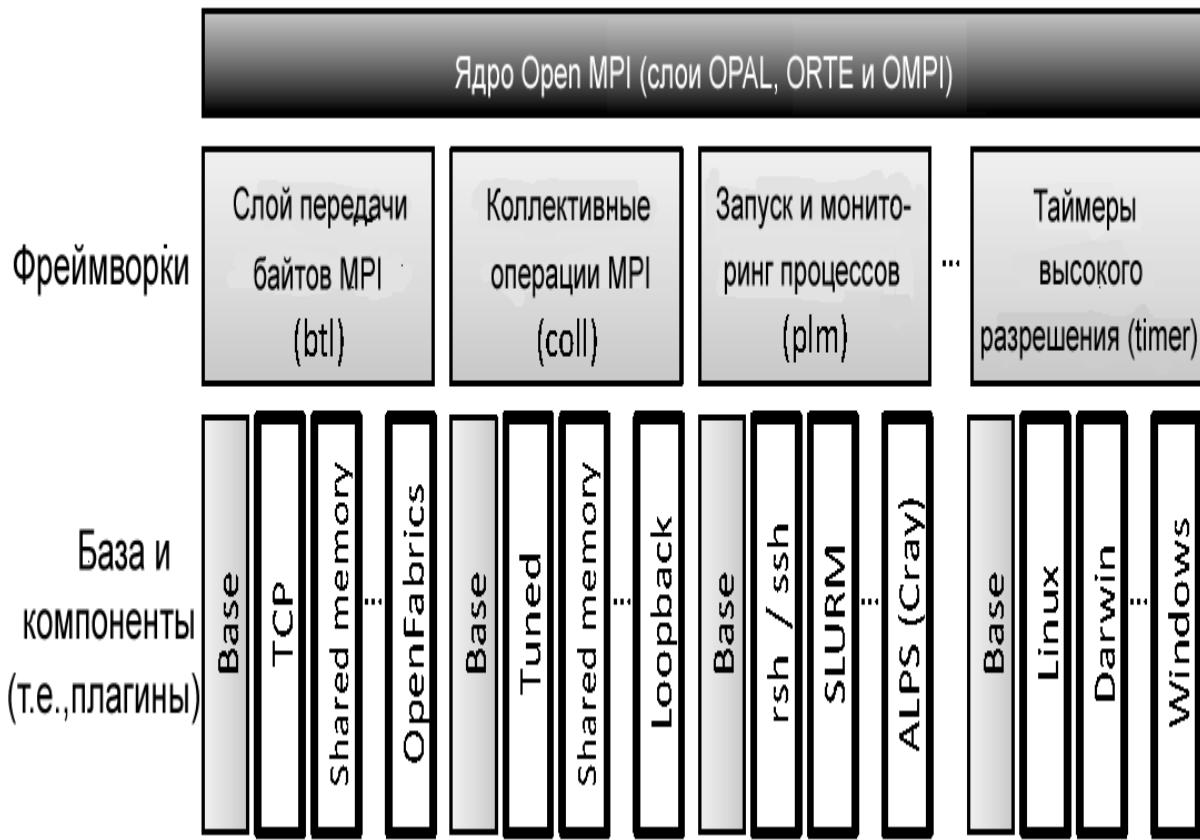


Рис.15.2: Архитектурное представление фреймворков в Open MPI — показаны всего лишь несколько фреймворков и компонентов из Open MPI (т.е., плагины). Каждый фреймворк содержит базовый код base и один или несколько компонентов. Эта структура реплицируется в каждом из слоев, показанных на рис.15.1. На данном рисунке показаны примеры фреймворков всех трех слоев: `btl` и `coll` относятся к слою OMPI, `plm` относится к слою ORTE, а `timer` относится к слою OPAL.

Наконец, еще одним важным преимуществом использования фреймворков и компонентов является присущее им свойство сочетаемости. Наличие в версии MPI v1.5 более чем 40 фреймворков предоставляет пользователям возможность по-разному соединять различные плагины различных типов и позволяет им создавать программный стек, который наиболее эффективен в их конкретной системе.

Фреймворки плагинов

Каждый фреймворк является полностью самодостаточным и находится в своем собственном каталоге в дереве исходного кода Open MPI. Имя подкаталога будет таким же, как и имя фреймворка; например, фреймворк `memory` находится в каталоге `memory`. В каталогах фреймворков имеются, по меньшей мере, следующие три составляющих:

1. *Определение интерфейса компонента:* Заголовочный файл с именем `<framework>.h` будет расположен в каталоге фреймворка верхнего уровня (например, в фреймворке `Memory` будет файл `memory/memory.h`). В этом всем известном заголовочном файле определяются интерфейсы, которые должны поддерживаться в каждом компоненте. В этом заголовке имеются указатели функций `typedef` для функций интерфейсов, структуры, предназначенные для работы с указателями с этими функциями, а также все другие необходимые типы, поля атрибутов, макросы, объявления и т.д.
2. *Базовый код:* В подкаталоге `base` находится связующий код, в котором реализован основной набор функций фреймворка. Например, базовым каталогом фреймворка `memory` будет каталог `memory/base`. К числу базовых обычно относятся функции, обеспечивающие функции

ционирование фреймворка, например, осуществляющие поиск и открытие компонентов во время выполнения фреймворка, а также утилиты общего назначения, которые могут использоваться несколькими компонентами и т.д.

3. *Компоненты*: Все другие подкаталоги в каталоге фреймворка считаются компонентами. Точно также, как и у фреймворка, имена компонентов являются именами подкаталогами (например, в подкаталоге `memory/posix` находится компонент POSIX фреймворка Memory).

Подобно тому, как в каждом фреймворке определяются интерфейсы, которые должны поддерживаться в его компонентах, в фреймворке также определяются другие особенности функционирования, например, как фреймворки будут загружаться, как будут выбираться используемые компоненты и как будет завершаться работа компонентов. Ниже приведены два примера различий фреймворков: фреймворки типа «несколько из нескольких» и типа «один из нескольких», а также статические и динамические фреймворки.

Фреймворки типа «несколько из нескольких»

Некоторые фреймворки обладают функциями, которые в одном и том же процессе можно реализовывать несколькими различными способами. Например, фреймворк сети типа «точка-точка» в проекте Open MPI будет загружать несколько плагинов драйверов для того, чтобы в одном процессе можно было отправлять и получать сообщения из сетей нескольких типов.

Такие фреймворки, как правило, открывают все компоненты, которые они могут найти, а затем спрашивают у каждого компонента, должны ли они работать. Компоненты, изучая систему, в которой они работают, определяют, должны ли они работать. Например, сетевой компонент типа «точка-точка» определит, если ли и активны ли в системе типы сетей, которые он поддерживает. Если их нет, то компонент ответит, что он не должен запускаться, в результате чего фреймворк закроет и выгрузит этот компонент. Если этот тип сети доступен, то компонент ответит, что он должен быть запущен, в результате чего фреймворк будет держать этот компонент открытым для дальнейшего использования.

Фреймворки типа «один из нескольких»

Другие фреймворки предоставляют функции, для которых не имеет смысла во время выполнения иметь более одной доступной реализации. Например, создание согласованной контрольной точку параллельно выполняемого задания, что значит, что задание может быть «заморожено» и его можно будет возобновить позже, т. е. оно должна использоваться одна и та же система фоновых контрольных точек. Плагин, который взаимодействует с нужной системой фоновых контрольных точек является единственным плагином контрольных точек, который должен загружаться в каждом процессе, а все остальные плагины - не нужны.

Динамические фреймворки

Большинство фреймворков позволяют с помощью разделяемых объектов DSO загружать свои компоненты во время выполнения. Это наиболее гибкий метод поиска и загрузки компонентов; он позволяет явно не указывать конкретные загружаемые компоненты, загружать компоненты сторонних производителей, не входящие в основной дистрибутив проекта Open MPI, и т.д.

Статические фреймворки

Некоторые компоненты типа «один из нескольких» имеют дополнительные ограничения, которые заставляют во время компиляции (а не во время выполнения) выбирать один и только один компонент из нескольких. Статическая компоновка компонентов типа «один из нескольких» позволяет напрямую вызывать функции-члены (и не использовать указатель на функцию), что может быть

важно для обеспечения высокой производительности. Одним из примеров является фреймворк `mempool`, который предоставляет реализации функции `mempool()`.

Кроме того, некоторые фреймворки предоставляют функции, которые, возможно, должны быть использованы еще до полной инициализации Open MPI. Например, использование некоторых сетевых стеков требуют сложных моделей регистрации памяти, что, в свою очередь, требует замены процедур управления памятью библиотеки языка C, используемой по умолчанию. Поскольку управление памятью влияет на весь процесс, замена стандартной схемы может быть выполнена только перед запуском основного модуля `main`. Поэтому такие компоненты должны быть статически скомпонованы с процессами в Open MPI, т. к. обращение к ним может происходить перед запуском модуля `main`, т. е. задолго до того, как будет инициализирован MPI.

Компоненты плагинов

Плагины Open MPI состоят из двух частей: структуры *компонент* и структуры *модуля*. Структура компонента и функций, к которым он обращается, как правило, вместе именуются как «компонент». Аналогичным образом собирательное понятие «модуль» относится к структуре модуля и к его функциям. Деление немного напоминает деление на классы и объекты в языке C++. В каждом процессе есть только один компонент; в нем описывается общий плагин с некоторыми полями, которые являются общими для всех компонентов (независимо от фреймворка). Если компонент выбирается для запуска, то он используется для создания одного или нескольких модулей, которые обычно выполняют основную часть функций, необходимых фреймворку.

На протяжении следующих нескольких разделов мы создадим структуры, необходимые для компонента TCP в фреймворке BTL (слой побайтовой передачи данных). Фреймворк BTL используется при передаче сообщений типа «точка-точка»; компонент TCP, что очевидно, использует TCP в качестве основного транспорта для передачи сообщений.

Структура компонента

Независимо от фреймворка в каждом компоненте есть всем известная статически выделяемая и инициализируемая структура компонента. Структура должна называться согласно шаблону как `mca_<framework>_<component>_component`. Например, структура драйвера сети TCP во фреймворке BTL называется `mca_btl_tcp_component`.

Наличие символов компонентов, созданных по шаблону, гарантирует, что между именами компонентов не будет никаких конфликтов, и позволяет ядру МСА искать структуру произвольного компонента при помощи `dlsym(2)` (или соответствующего эквивалента в каждой поддерживаемой операционной системе).

Структура базового компонента содержит некоторую информацию об используемых ресурсах, например, официальное название компонента, версия, принадлежность версии фреймворка и т.д. Эти данные используются для отладки, учета и во время выполнения для проверки соблюдения совместимости.

```
struct mca_base_component_2_0_0_t {
    /* Номер версии структуры компонента */
    int mca_major_version, mca_minor_version, mca_release_version;

    /* Стока с именем фреймворка, к которому принадлежит компонент,
       и версия API фреймворка, к которому принадлежит данный компонент */
    char mca_type_name[MCA_BASE_MAX_TYPE_NAME_LEN + 1];
    int mca_type_major_version, mca_type_minor_version,
        mca_type_release_version;

    /* Имя и номер версии компонента */
}
```

```

char mca_component_name[MCA_BASE_MAX_COMPONENT_NAME_LEN + 1];
int mca_component_major_version, mca_component_minor_version,
    mca_component_release_version;

/* Указатели на функции */
mca_base_open_component_1_0_0_fn_t mca_open_component;
mca_base_close_component_1_0_0_fn_t mca_close_component;
mca_base_query_component_2_0_0_fn_t mca_query_component;
mca_base_register_component_params_2_0_0_fn_t
    mca_register_component_params;
};

}

```

Структура базового компонента является основой компонента TCP BTL; она содержит указатели на следующие функции:

- *Open.* Вызов функции *open* является инициализирующей функцией, к которой обращается компонент. Она позволяет компоненту инициализировать состояние самого компонента, проанализировать систему, в которой он работает, и определить, должен ли он выполняться. Если компонент должен выполняться всегда, он может в качестве указателя на функцию *open* передавать значение NULL.

Функция *open* компонента TCP BTL, как правило, инициализирует некоторые структуры данных и обеспечивает, чтобы пользователь не смог установить недопустимые параметры.

- *Close.* Когда фреймворк решает, что компонент не больше не нужен, он вызывает функцию *close*, которая позволяет компоненту освободить все ресурсы, которые для него были выделены. Когда процессы останавливаются, то для всех остальных компонентов также вызывается функция *close*. Однако, функция *close* также может быть вызвана для компонентов, запуск которых был отклонен во время выполнения, так что они могут быть закрыты и игнорироваться в течение всего процесса.

Функция *close* компонента TCP BTL закрывает прослушиваемые сокеты и освобождает ресурсы (например, приемные буферы).

- *Query.* Этот вызов является обобщением функции, запрашивающей необходимость запуска компонента. Этот специальный вызов используется не во всех фреймворках — в некоторых требуется более специализированная функция запроса.

Во фреймворке BTL обобщенная функция *query* не используется (в нем определяется своя собственная функция; смотрите ниже), поэтому TCP BTL ее не заполняет.

- *Регистрация параметров.* Эта функция, как правило, является первой функцией, которая вызывается в компоненте. Она позволяет компоненту зарегистрировать соответствующие параметры времени выполнения, которые можете задавать пользователь. Параметры времени выполнения будут рассмотрены ниже.

Функция *register* компонента TCP BTL создает различные параметры времени выполнения, которые может устанавливать пользователь, например, один из них позволяет пользователю указывать, какой интерфейс IP будет использоваться.

Компонентная структура также может быть расширена в каждом конкретном фреймворке и/или в каждом конкретном базисном коде. Во фреймворке обычно создают новую структуру компонента с базовой структурой компонента в качестве первого элемента. Такая вложенность позволяет фреймворкам добавлять свои собственные атрибуты и указателей на функции. Например, для фреймворка, для которого требуется более специализированная функция запроса (в сравнении с

функцией *query*, которая реализована в базовом компоненте), можно добавить указатель на функцию внутри структуры, специализированной под конкретный фреймворк.

Эта методика используется во фреймворке *btl* в MPI, в котором реализуются функции MPI для передачи сообщений типа «точка-точка».

```
struct mca_btl_base_component_2_0_0_t {
    /* Структура базового компонента */
    mca_base_component_t btl_version;
    /* Блок данных базового компонента */
    mca_base_component_data_t btl_data;

    /* Функции query, специальные для фреймворка btl */
    mca_btl_base_component_init_fn_t btl_init;
    mca_btl_base_component_progress_fn_t btl_progress;
};

};
```

Например, функции *query* фреймворка TCP BTL и функция *btl_init* компонента TCP BTL выполняют следующее:

- Создается прослушиваемый сокет для каждого интерфейса IPv4 и IPv6, идущего вверх.
- Создается модуль для каждого интерфейса IP, идущего вверх.
- В центральном репозитории регистрируется кортеж (*IP address, port*), т. е. IP адрес и порт для каждого интерфейса IP, идущего вверх, с тем, чтобы в других процессах MPI было известно, как связаться с данным процессом.

Аналогичным образом плагины могут расширять структуру компонента конкретного фреймворка, добавляя в нее свои собственные элементы. Это делает компонент *tcp* во фреймворке *btl*; он кэширует многие члены-данные в своей собственной структуре компонента.

```
struct mca_btl_tcp_component_t {
    /* Структура компонента, специальная для фреймворка btl */
    mca_btl_base_component_2_0_0_t super;

    /* Некоторые данные-члены, специальные для компонента TCP BTL */
    /* Количество интерфейсов TCP на данном сервере */
    uint32_t tcp_addr_count;

    /* Дескриптор сокета, слушающего IPv4 */
    int tcp_listen_sd;

    /* ... и многое другое, что здесь не показано */
};
```

Такая методика вложенных структур является эффективной и простой имитацией одиночного наследования языка C++: указатель на экземпляр структуры *struct mca_btl_tcp_component_t* может быть приведен к любому из трех типов, т. е. он может использоваться на уровне абстракций, на котором непонятны «производные» типы.

Надо сказать, что такое приведение типов, как правило, не одобряется в Open MPI, поскольку оно может привести к невероятно тонким, трудно обнаруживаемым ошибкам. Исключение может быть сделано для этого варианта эмуляции C++, поскольку в этом случае задается строго определенное поведение, которое помогает соблюдать границы абстракций.

Структура модуля

Структуры модулей определяются индивидуально в каждом фреймворке; между ними мало общего. В зависимости от того, какой используется фреймворк, компонент создает один или несколько экземпляров модулей и укажет, что они должны использоваться.

Например, во фреймворке BTL, один модуль обычно соответствует одному сетевому устройству. Если процесс MPI работает на Linux сервере с тремя устройствами Ethernet, то компонент TCP BTL создаст три модуля TCP BTL; один модуль соответствует каждому устройству Linux Ethernet. Затем каждый из модулей будет полностью ответственен за отправку и получение всех данных через конкретное сетевое устройство.

Объединяем все вместе

На рис.15.3 показана вложенность структур в компоненте TCP BTL и то, как он генерирует по одному модулю для каждой из трех устройств Ethernet.

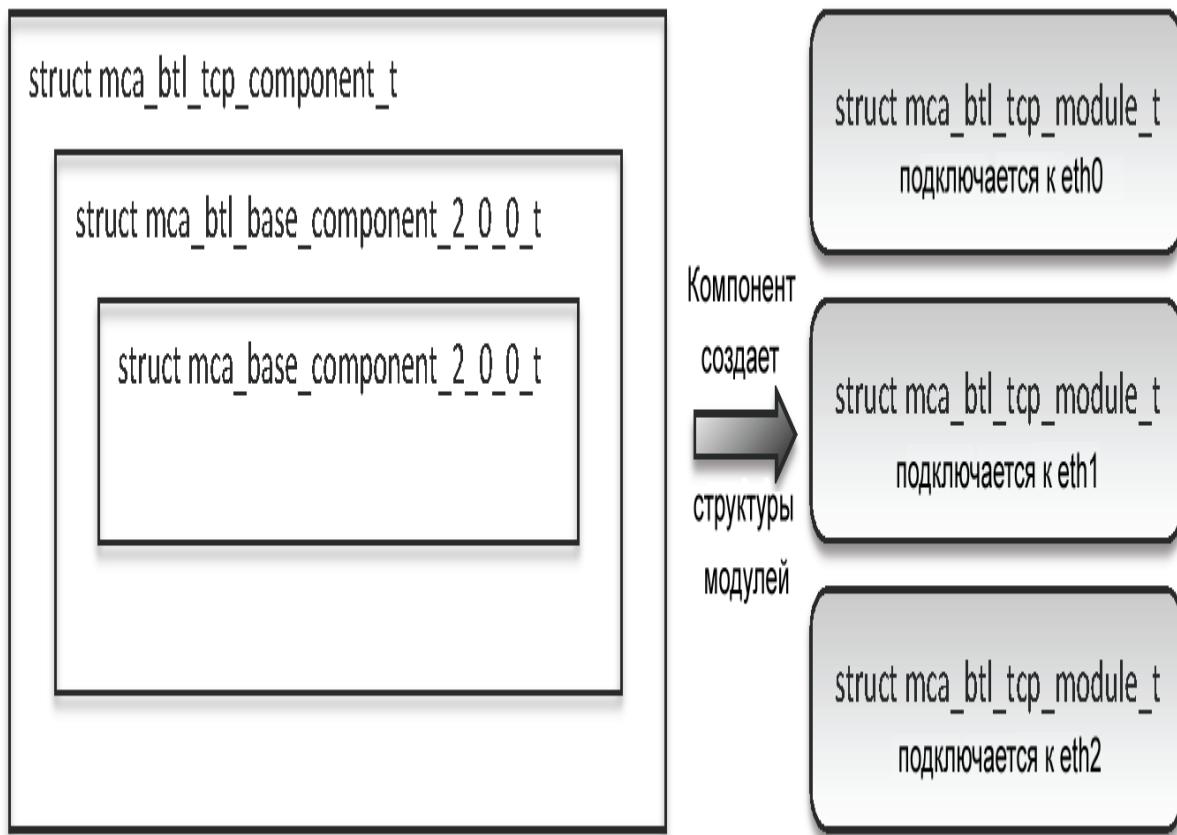


Рис.15.3: С левой стороны показана вложенность структур в компоненте TCP BTL. Справа показано, как компонент генерирует по одному модулю для каждого интерфейса Ethernet, идущего вверх.

Композиция модулей BTL, таким образом, позволяет движку верхнего уровня MPI обрабатывать все сетевые устройства одинаковым образом и выполнять привязку к каналу пользовательского уровня.

Например, рассмотрим отправку большого сообщения с помощью конфигурации из трех устройств, описанных выше. Предположим, что для того, чтобы достичь предполагаемого получателя, можно использовать любое из трех устройств Ethernet (достижимость определяется сетями TCP, масками и некоторыми строго задаваемыми эвристиками). В данном случае отправитель разделит большое сообщение на множество фрагментов. Каждый фрагмент будет назначен в цикле одному из модулей TCP BTL (поэтому каждому модулю будет назначено примерно одна треть

фрагментов). Затем каждый модуль отправляет назначенные ему фрагменты через его собственное соответствующее устройство Ethernet.

Эта схема может показаться сложной, но она удивительно эффективна. За счет того, что пересылка большого сообщения происходит с помощью конвейера через несколько модулей TCP BTL, типичная среда высокопроизводительных вычислений (например, когда каждое устройство Ethernet находится на отдельнойшине PCI) может через несколько устройств Ethernet поддерживать почти максимальную пропускную скорость.

Параметры времени выполнения

Разработчики при написании кода часто принимают решения, например, следующие:

- Должен ли использоваться алгоритм А или алгоритм В?
- Буфер какого размера должен предварительно выделяться?
- Насколько долгим должен быть таймаут?
- На какой размер сообщений должен быть настроен сетевой протокол?
- ... и так далее.

Пользователи склонны полагать, что разработчики ответят на подобные вопросы так, что это, в общем случае, подойдет для большинства типов систем. Тем не менее, в сообществе, связанном с высокопроизводительными вычислениями, много ученых и инженеров - опытных пользователей, которые хотят для каждого возможного варианта вычислительного цикла активно настраивать свои аппаратные и программные стеки. Хотя эти пользователи обычно не хотят возиться с фактическим кодом собственной реализации MPI, им интересно в различных обстоятельствах возиться с выбором различных внутренних алгоритмов, выбором различных моделей потребления ресурсов или принудительно задавать конкретные сетевые протоколы.

Поэтому когда разрабатывался проект Open MPI, была добавлена система параметров MCA; система представляет собой гибкий механизм, позволяющий пользователям во время исполнения изменять значения внутренних параметров Open MPI. В частности, разработчики могут везде в базовом коде Open MPI регистрировать строковые и целочисленные параметры MCA, указывающее соответствующее значение, используемое по умолчанию, и строку описания, определяющую, что это за параметр и как он используется. Общее правило состоит в том, что разработчики вместо того, чтобы жестко кодировать константы, используют параметры MCA, которые устанавливаются во время выполнения, что позволяет опытным пользователям настраивать то, как система будет себя вести на этапе выполнения.

В базовом коде трех абстрактных слоев есть ряд параметров MCA, но основная часть параметров MCA системы Open MPI размещены в отдельных компонентах. Например, в плагине TCL BTL есть параметр, определяющий должен ли использоваться только интерфейсы TCPv4, только интерфейсы TCPv6 или оба типа интерфейсов. Кроме того, с помощью еще одного параметра TCP BTL можно точно указывать какие используются устройства Ethernet.

Пользователи могут узнать, какие параметры доступны, с помощью специального, предназначенного для пользователей инструментального средства (`ompi_info`), работающего из командной строки. Значения параметров можно устанавливать несколькими способами: в командной строке, через переменные среды окружения, через реестр Windows, или с помощью системных или пользовательских файлов в стиле INI.

Система параметров MCA дополнила идею гибкого выбора плагинов во время выполнения и оказалось весьма ценной для пользователей. Хотя разработчики Open MPI старались выбирать разумные значения, используемые по умолчанию в самых разнообразных ситуациях, каждая высокопроизводительная среда имеет свои отличия. Неизбежно существуют среды, для которых не под-

ходят значения параметров, задаваемые в Open MPI по умолчанию, и которые, возможно, даже вредят поддержке высокой производительности. Система параметров МСА позволяет пользователям быть активными и настраивать поведение Open MPI в соответствие с их средой. Это не только упрощает ситуацию с запросами об изменениях в Open MPI и/или с сообщениями об ошибках, но также позволяет пользователям экспериментировать с пространством параметров и находить лучшую конфигурацию для их конкретной системы.

15.3. Усвоенные уроки

Неизбежно, что при наличии такой разношерстной группы основных разработчиков Open MPI, мы должны были каждый раз что-то изучать, и что, как группа, мы должны были многому научиться. Ниже перечислены лишь некоторые из этих уроков.

Производительность

Производительность при передачи сообщений и использование ресурсов являются королем и королевой высокопроизводительных вычислений. Open MPI был специально разработан таким образом, чтобы мог работать на самом переднем крае высокой производительности: невероятно низкие задержки при отправке коротких сообщений, чрезвычайно высокая скорость добавления коротких сообщений в поддерживаемые сети, быстрое достижение максимальной пропускной способности для больших сообщений и т.д. Абстракция является хорошим делом (по многим причинам), но она должна разрабатываться с осторожностью с тем, чтобы она не ухудшала производительность. Или, иначе говоря: тщательно выбирайте абстракции, которые сами понемногу ухудшают производительность стеков вызовов (в сравнении со стеками вызовов с использованием API).

Т.е. также должно признать, что в некоторых случаях следует выбрасывать абстракцию, а не архитектурное решение. Показательный пример: в Open MPI есть фрагменты ассемблерного кода, закодированные вручную, для некоторых из наиболее критичных к производительности операций, например, блокировка совместно используемой памяти и атомарные операции.

Стоит отметить, что на рис.15.1 и 15.2 показаны два различных варианта архитектуры Open MPI. В них не представлены стеки вызовов времени выполнения или обращение к слою вызовов для разделов кода, где нужна высокая производительность.

Усвоенный урок:

Допускается (хотя и нежелательно) и, к сожалению, иногда необходимо иметь объемистый и сложный код для достижения высокой производительности (например, вышеупомянутый ассемблерный код). Тем не менее, всегда предпочтительнее потратить время, пытаясь выяснить, как создать хорошие абстракции с тем, чтобы по мере возможности дискретизировать и скрыть сложность. Несколько недель проектирования могут сэкономить сотни или тысячи часов у разработчиков, которые им потребуются на поддержку запутанного непонятного кода, похожего на спагетти.

Стоя на плечах гигантов

Мы в Open MPI активно пытались избежать заново изобретать код, который кто-то уже написал (если такой код был совместим с лицензией BSD, используемой в Open MPI). В частности, у нас нет никаких угрызений совести относительно непосредственного повторного использования или взаимодействия с чужим кодом.

Когда делается попытка решить очень сложные технические задачи, то не место придерживаться принципа «изобретено не здесь»; единственное, что имеет смысл, всякий раз, когда это возможно, повторно использовать внешний код. Такое повторное использование кода позволяет разработчи-

кам сосредоточиться на проблемах, которые уникальны для проекта Open MPI; нет смысла повторно решать проблему, которая кем-то уже решена.

Хорошим примером такого повторного использования кода является пакет GNU Libtool Libltdl. Libltdl это небольшая библиотека, которая предоставляет переносимый интерфейс API для открытых объектов DSO и поиска в них символов. Libltdl поддерживается в самых различных операционных системах и средах, в том числе и в Microsoft Windows.

В Open MPI можно было реализовать эти функциональные возможности самостоятельно, но — зачем? Libltdl является прекрасным образцом программного обеспечения, которое активно поддерживается, совместимо с лицензией Open MPI и предоставляет именно те функциональные возможности, которые были необходимы. Учитывая все это, разработчики Open MPI не получат каких-либо реальных выгод, если заново напишут эти функции.

Усвоенный урок:

Если где-нибудь есть подходящий решение, то не стесняйтесь и воспользуйтесь им и не тратьте время, пытаясь его повторно повторить.

Оптимизация обычно выполняемых операций

Еще один направляющий архитектурный принцип состоит в том, чтобы оптимизировать наиболее часто выполняемые операции. Например, ударение делается на разделение многих операций на две части: на настройку и многократно выполняемое действие. Предполагается, что настройка может оказаться затратной (что означает: медленной). Так что сделаем ее *один* раз, и покончим с ней. Оптимизируем гораздо более распространенный случай: повторно выполняемую операцию.

Например, функция `malloc()` может быть медленной, особенно если страницы памяти должны выделяться операционной системой. Поэтому вместо того, чтобы выделить то количество байтов, которое необходимо для одного входящего сетевого сообщения, выделяется место, достаточное сразу для *группы* входящих сообщений, которое затем делится на буферы отдельных сообщений, и создается список свободных блоков памяти, который поддерживает их использование. Таким образом, первый запрос буфера для сообщения может быть медленным, но *следующие* запросы будут выполняться гораздо быстрее, поскольку они будут лишь удалением буферов из очереди свободных блоков.

Усвоенный урок:

Разбиваем обычные операций (по крайней мере) на две части: настройка и повторяющееся действие. Мало того, что код будет работать лучше, его, может быть, будет проще поддерживать в течение долгого времени, поскольку различные действия разделены.

Прочие уроки

Было усвоено слишком много других уроков с тем, чтобы их можно было бы здесь подробно описать; приведем еще несколько уроков, которые можно суммировать следующим образом:

- Нам повезло опереться на более чем 15 летний опыт исследований в области высокопроизводительных систем и еще в течение более восьми лет создавать проекты, которые были (в основном) успешными. Приступая к новому проекту программного обеспечения, *посмотрите в прошлое*. Удостоверьтесь в том, что понимаете, что уже было сделано, почему это было сделано и каковы были сильные и слабые стороны сделанного.

- Концепция компонентов, которая допускает несколько различных реализаций одной и той же функциональности, спасала нас много раз, причем как технически, так и политически. Плагины — это дело хорошее.
- Кроме того, мы постоянно добавляли и удаляли фреймворки по мере необходимости. Когда разработчики начинают спорить о «правильном» способе реализации новой возможности, то добавляется фреймворк, который экранирует компоненты, реализующие эту возможность. Или когда приходят новые идеи, из-за которых становятся ненужными устаревшие фреймворки, то не стесняйтесь и удаляйте такие обертки.

Заключение

Если бы нам пришлось перечислить три наиболее важных факта, с которыми мы познакомились в проекте Open MPI, я думаю, что это бы выглядело следующим образом:

- Одно и то же значение не подходит для всех (пользователей). Плагин времени выполнения и сопутствующая ему система параметров МСА предоставляют пользователям возможность гибкой настройки, что необходимо в мире переносимого программного обеспечения. Сложные системы программного обеспечения не могут (каждый раз) волшебным образом адаптироваться к конкретной системе; предоставление средств управления пользовательского уровня позволяет человеку понять и перенастроить, если программа ведет себя субоптимально.
- Различия хороши. Разногласия между разработчиками также хорошая вещь. Охват проблем ведет к статус-кво; самодовольству не место. Фраза смелого аспиранта: "Давай, проверим это ..." может привести к возникновению базы для абсолютно новой возможности или к существенному усовершенствованию изделия.
- Хотя это выходит за рамки данной книги, но важное место занимают люди и сообщество. Важное место.

16. OSCAR

Глава 16 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

С момента внедрения системы EMR (системы электронных медицинских записей) были предназначены для осуществления связи между физическими и цифровыми мирами обслуживания пациентов. Правительства стран всего мира пытались предоставить решение, которое позволяло бы лучшим образом обслуживать пациентов при небольшой цене, сокращая объем бумажной документации, которая обычно создается при работе медицинских учреждений. Многие правительства успешно справились с задачей создания такой системы - некоторые, такие, как правительство провинции Онтарио Канады не справились с ней (достаточно вспомнить так называемый "скандал с электронной системой здравоохранения eHealth" в Онтарио, который, судя по отчету Генеральной комиссии по аудиту, обошелся для налогоплательщиков в сумму 1 миллион канадских долларов).

Система EMR позволяет перевести в электронный вид список пациентов и, при корректном использовании, должна упрощать процесс обслуживания пациентов медицинскими работниками. Качественная система должна предоставлять медицинскому работнику возможность точной оценки текущего и последующего состояний пациента, его истории болезни, результатов лабораторных исследований, истории прошлых посещений, и.т.д.

OSCAR (Open Source Clinical Application Resource - программное обеспечение для медицинских учреждений с открытым исходным кодом) является проектом с практически десятилетней историей, созданным в университете McMaster, Гамильтон, Канада и направленным на формирование сообщества вокруг приложения с открытым исходным кодом, работающего с целью передачи в

распоряжение медицинских работников описанной системы по низкой цене или вообще бесплатно.

В составе OSCAR имеется ряд подсистем, которые реализуют функции на основе каждого из компонентов системы. Например, компонент oscarEncounter предоставляет интерфейс для прямого взаимодействия с картой пациента; компонент Rx3 является модулем директив, автоматически проверяющим наличие аллергических реакций и непереносимости медицинских препаратов и позволяющим медицинскому работнику отправить предписание по факсу в аптеку непосредственно с помощью пользовательского интерфейса; компонент Integrator позволяет осуществлять обмен данными между несколькими совместимыми друг с другом версиями систем EMR. Все эти отдельные компоненты объединены для формирования стандартной системы для взаимодействия с пользователями OSCAR.

Система OSCAR может не подходить для каждого медицинского работника; например, не все функции системы могут оказаться полезными для специалиста, причем система не может быть просто настроена. Однако, она предоставляет завершенный набор возможностей для повседневного обслуживания пациентов среднестатистическим медицинским работником.

В дополнение к этому система OSCAR прошла сертификацию CMS 3.0 (и была принята к сертификации CMS 4.0), что позволяет медицинским работникам получать спонсорскую помощь в случае установки системы в своей клинике ([обратитесь к вебсайту "EMR Advisor"](#) в том случае, если вас интересуют подробности). Сертификат CMS может быть получен после успешной проверки выполнения ряда требований правительства Онтарио и оплаты взноса.

В данной главе мы в общих словах обсудим архитектуру системы OSCAR, описывая ее иерархию, основные компоненты и, что наиболее важно, воздействие на процесс развития проекта решений, принятых в прошлом. В качестве заключения мы обсудим то, как система OSCAR могла быть спроектирована на сегодняшний день в том случае, если бы у нас была возможность заняться этим.

16.1. Системная иерархия

Являясь веб-приложением Tomcat, система OSCAR по большей части следует шаблону проектирования "модель-представление-контроллер" (MVC). Это значит, что код модели (объекты доступа к данным - Data Access Objects, или DAO) отделен от кода контроллера (сервлетов), а их код, в свою очередь, отделен от представлений (генерируемых с использованием технологии Java Server Pages или JSP). Наиболее важным различием между контроллером и представлениями является то, что сервлеты являются классами, а с помощью технологии JSP формируются HTML-страницы, разметка которых производится с помощью кода на языке Java. Данные размещаются в памяти во время выполнения сервлета и при использовании технологии JSP производится чтение этих же данных, обычно путем чтения и записи атрибутов объекта обработки запроса. Практически любая страница, созданная с использованием технологии JSP в рамках системы OSCAR, спроектирован таким же образом.

16.2. Принятые в прошлом решения

Я упоминала о том, что OSCAR является достаточно взрослым проектом. Это обстоятельство оказалось воздействие на эффективность применения в рамках проекта шаблона проектирования MVC. Если говорить кратко, существуют участки кода, в которых этот шаблон проектирования вообще не используется ввиду того, что они были разработаны до момента начала активного применения шаблона проектирования MVC. Некоторые из наиболее часто используемых возможностей реализованы именно таким образом; например, выполнение множества действий с демографическими данными (записями пациентов) осуществляется в рамках файла исходного кода

`demographiccontrol.jsp` - эти действия включают создание записей пациентов и обновление их данных.

Возраст системы OSCAR является препятствием для решения множества проблем, затрагивающих дерево исходного кода на сегодняшний день. На самом деле были приложены значительные усилия для улучшения ситуации, среди которых принудительное использование правил проектирования в ходе процесса обзора исходного кода. Этот подход, выбранный сообществом на данный момент, служит для повышения качества процесса взаимодействия в будущем и предотвращения попадания некачественного исходного кода в кодовую базу проекта, что было проблемой в прошлом.

Этот подход ни в коем случае не является ограничением того, как мы могли бы проектировать части системы сегодня; однако, этот подход усложняет процесс принятия решений при исправлении ошибок в устаревших частях системы OSCAR. Если вам или кому-то другому предстоит исправить ошибку в функции создания записи пациента, будете ли вы исправлять ошибку, используя тот же стиль, что был применен при создании существующего кода? Или все-таки вы заново полностью разработаете модуль, точно следя за шаблоном проектирования MVC?

Являясь разработчиками, мы должны трепетно взвешивать наши возможности в подобных ситуациях. Не существует гарантии того, что если вы повторно спроектируете часть системы, не будет допущено новых ошибок и в том случае, когда производится работа с реальными данными пациентов, решение должно приниматься чрезвычайно аккуратно.

16.3. Управление версиями

Большую часть периода существования проекта OSCAR для управления деревом исходного кода использовалась система контроля версий CVS. Вносимые изменения обычно не проверялись на корректность, поэтому имелась возможность добавить в репозиторий код, который мог привести к невозможности сборки. Разработчикам было сложно отслеживать изменения, особенно в случае присоединения к команде новых разработчиков на поздних этапах жизненного цикла проекта. Новый разработчик мог увидеть что-либо, что он желал бы изменить, внести изменения и отправить их в ветку исходного кода за несколько недель до того, как кто-либо заметит значительные модификации (данная ситуация особенно актуальна в период длительных праздников, таких, как Рождественские каникулы, в течение которых очень малое количество людей исследует дерево исходного кода).

Но положение вещей изменилось: дерево исходного кода проекта OSCAR на сегодняшний день находится под управлением системы контроля версий git. Любые модификации кода из основной ветки должны преодолеть проверку стиля кода и модульное тестирование, быть успешно скомпилированы и проверены разработчиками. (Большая часть этой работы выполняется с помощью комбинации сервера системы непрерывной интеграции [Hudson](#) и инструмента проверки стиля исходного кода [Gerrit](#).) Управление проектом стало более надежным. Многие проблемы, вызванные некорректной работой с деревом исходного кода проекта, были решены.

16.4. Модели данных/DAO

При изучении дерева исходного кода проекта OSCAR вы можете заметить, что существует множество различных способов осуществления доступа к базе данных: вы можете использовать прямое соединение с базой данных с помощью класса с именем `DBHandler`, устаревшее соединение с использованием модели Hibernate или модель JPA общего назначения. По мере появления новых и более простых моделей взаимодействия с базой данных, они интегрируются в состав проекта OSCAR. В результате на данный момент процесс взаимодействия системы OSCAR с данными из

базы данных MySQL является не достаточно очевидным и различия между тремя описанными методами доступа к данным могут быть описаны лучшим образом с помощью примеров.

EForms (DBHandler)

Система EForm позволяет пользователям создавать свои формы для привязки их к записям пациентов - эта возможность обычно используется для замены бумажных бланков на их цифровые версии. При каждом создании формы определенного типа загружается шаблон формы из файла; после этого данные формы сохраняются в базе данных для каждого ее экземпляра. Каждый экземпляр формы привязывается к записи пациента.

Система EForms позволяет вам запрашивать определенные типы данных из списка пациентов или другой области данных системы с использованием SQL-запросов в свободной форме (которые заданы в файле с именем `apconfig.xml`). Это может быть очень полезным, так как форма может быть загружена, после чего немедленно заполнена демографическими данными или другой соответствующей информацией без вмешательства пользователя; например, вам не придется вписывать имя пациента, его возраст, дату рождения, место рождения, номер телефона или последнюю медицинскую запись при работе с конкретным пациентом.

При начальном проектировании модуля EForm было принято архитектурное решение, заключающееся в использовании необрабатываемых запросов к базе данных для заполнения POJO (простого Java-объекта в старом стиле - plain-old Java object) с именем `EForm` в контроллере, который впоследствии передается на уровень представления для вывода данных на экран так, как это сделано с `JavaBean`. Использование объекта POJO в данном случае приближает архитектурное решение к решениям `Hibernate` или `JPA`, о которых я расскажу в следующих разделах.

Все функции, относящиеся к сохранению экземпляров класса `EForm` и шаблонов, осуществляются с помощью необрабатываемых SQL-запросов, выполняемых классом `DBHandler`. В конечном счете, класс `DBHandler` является оберткой над простым объектом JDBC и не проводит исследование запроса перед его отправкой серверу SQL. Следует добавить, что использование класса `DBHandler` является потенциальной угрозой безопасности, так как он позволяет отправлять серверу непроверенные SQL-запросы. Любой класс, использующий класс `DBHandler`, должен реализовывать свои собственные алгоритмы проверки для того, чтобы быть уверенным в неосуществимости SQL-инъекций.

В зависимости от типа приложения, которое вы разрабатываете, прямой доступ к базе данных иногда может оказаться подходящим решением. В определенных случаях такая возможность позволяет даже повысить скорость разработки. Использование этого метода для доступа к базе данных не соответствует шаблону проектирования "модель-представление-контроллер", хотя в том случае, если хотите изменить структуру вашей базы данных (модель), вам придется изменить SQL-запрос в другом месте (в контроллере). Иногда добавление определенных столбцов или изменение их типов в таблицах базы данных системы OSCAR требует выполнения подобной процедуры вмешательства всего лишь для реализации простейших возможностей.

Вас может не удивлять тот факт, что объект `DBHandler` описан в одной из старейших частей исходного кода и все еще является нетронутым. Лично я не знаю, где он возник, но я предполагаю, что этот класс является наиболее "примитивным" типом класса для доступа к базе данных в рамках дерева исходного кода системы OSCAR. Новый исходный код не может использовать этот класс, а в том случае, если использующий его исходный код все же попытается добавить в репозиторий, он будет автоматически отклонен.

Демографические записи (Hibernate)

Демографическая запись содержит основные метаданные, имеющие отношение к пациенту: например, его имя, возраст, адрес, родной язык и пол; будем считать, что эти данные появляются после заполнения пациентом формы приема в ходе его первого визита к врачу. Все эти данные извлекаются и выводятся в форме части мастер-записи системы OSCAR (OSCAR's Master Record) для определенной демографической записи.

Использование Hibernate для осуществления доступа к базе данных значительно безопаснее использования класса DBHandler. С одной стороны вам приходится четко указывать то, какие столбцы соответствуют каким полям вашего объекта модели (в данном случае, класса Demographic). Если вы хотите выполнить сложные объединения запросов, они могут быть осуществлены с использованием заранее подготовленных объявлений. Наконец, вы получите объект исключительно того типа, который вы описывали при осуществлении запроса, что очень удобно.

Процесс работы с парами объектов доступа к данным (DAO) и моделей при использовании Hibernate достаточно прост. В случае объекта Demographic существует файл с именем `Demographic.hbm.xml`, который описывает связи между полями объекта и столбцами таблицы базы данных. Файл описывает то, к какой таблице следует обратиться, а также какой тип объекта следует вернуть. При запуске системы OSCAR данный файл должен быть прочитан, после чего должна быть проведена проверка достоверности прочитанных данных, предназначенная для того, чтобы быть уверенным в реальной возможности создания описанного типа связей (в случае неудачной проверки процесс запуска сервера прерывается). В процессе работы сервера вы можете создать экземпляр класса `DemographicDao` и выполнять запросы с помощью него.

Преимуществом использования Hibernate по сравнению с DBHandler является то, что все запросы к серверу базы данных осуществляются с использованием заранее подготовленных объявлений. Это обстоятельство ограничивает возможность свободного выполнения SQL-запросов в процессе работы системы, но при этом также предотвращает любые типы атак на основе SQL-инъекций. Hibernate всегда будет формировать сложные запросы для выборки данных, причем запросы не всегда формируются чрезвычайно эффективным способом.

В предыдущем разделе я упомянула о примере модуля EForm, использующего класс DBHandler для заполнения объекта POJO. Это еще один логический шаг, направленный на предотвращение разработки подобного кода. В случае изменения модели придется изменить только файл с расширением `.hbm.xml` и класс модели (добавить новое поле и новые функции для получения/установки значений в новом столбце), причем эти действия не затронут остальных частей приложения.

Хотя метод работы с Hibernate и современное метода обращения к базе данных посредством класса DBHandler, он начинает устаревать. Его не всегда удобно использовать, а также он требует файла конфигурации большого размера для каждой таблицы базы данных, доступ к которой вы хотите получить. Добавление новой пары объектов занимает время и в том случае, если вы выполните эту операцию некорректно, система OSCAR даже не начнет работу. По этой причине в любом случае никто не должен разрабатывать новый код, использующий Hibernate напрямую. Для замены описанной технологии на новом этапе разработки была предложена технология JPA.

Интегратор согласования (JPA)

Новейший метод доступа к базе данных заключается в использовании стандартного API для долговременного хранилища данных Java (Java Persistent API - JPA). В том случае, если бы в рамках проекта OSCAR было принято решение о переходе от использования Hibernate к использованию другого соответствующего стандарту JPA для объектов DAO и моделей API для доступа к базе данных, процесс миграции стал бы проще. К сожалению, так как подобные технологии являются слишком "новыми" для проекта OSCAR, практически не существует частей системы, которые фактически используют предлагаемый метод для получения данных.

В любом случае, позвольте мне дать пояснения относительно данного метода. Вместо файла с расширением .hbm.xml вы можете добавлять аннотации к вашим объектам модели и DAO. Эти аннотации описывают таблицу базы данных, к которой следует обратиться, связи между полями объекта и столбцами базы данных, а также методы объединения запросов. Вся информация хранится в двух файлах и для работы больше ничего не требуется. На заднем плане все еще функционирует Hibernate, выполняя функции фактического извлечения данных из базы.

Все модели интегратора создаются с использованием функций JPA являются как замечательными примерами нового стиля доступа к базе данных, так и демонстрацией метода реализации новой технологии в рамках системы OSCAR. Данная модель все еще не используется во многих местах системы. Интегратор является относительно новым дополнением к исходному коду. Поэтому есть смысл использования этой новой модели доступа к данным вместо прямого использования Hibernate.

Затронем ставшую привычной в этой части раздела тему об аннотациях объектов POJO, которые используются в рамках JPA для реализации отлаженного процесса обработки данных. Например, во время процесса сборки интегратора создается файл SQL, который позволяет вам создать все необходимые таблицы базы данных - это чрезвычайно полезная возможность. Благодаря этой функции становится невозможным создание не соответствующих друг другу таблиц и объектов моделей (что вы можете сделать при использовании любого другого типа метода доступа к базе данных) и вам никогда не придется беспокоиться о именовании столбцов и таблиц. Прямые SQL-запросы не осуществляются, поэтому невозможно провести атаки на основе SQL-инъекций. Одним словом, этот метод "просто работает".

Принцип работы JPA может рассматриваться как аналогичный принципу работы системы ActiveRecord из состава фреймворка Ruby on Rails. Класс модели описывает типы данных, а также базу данных, которая хранит их; то же, что происходит с ними - добавление и извлечение - не должно волновать пользователя.

Недостатки Hibernate и JPA

Технологии Hibernate и JPA предоставляют некоторые преимущества при их стандартном использовании. В том случае, когда они используются для простого получения и сохранения данных, они позволяют значительно сократить время, затрачиваемое на разработку и отладку приложения.

Однако, это не означает, что их реализация в рамках системы OSCAR идеальна. Так как пользователь не описывает SQL-запросы, используемые при взаимодействии с базой данных для заполнения объекта POJO, соответствующего определенной строке, Hibernate предоставляет выбор лучшего способа осуществления данной операции. "Лучший способ" может быть представлен несколькими вариантами: Hibernate может выбрать метод простого извлечения данных строки или выполнить объединение запросов и получить большой объем информации единовременно. Иногда механизм объединения запросов может выходить из-под контроля.

Еще один пример: таблица casemgmt_note содержит все записи пациентов. Каждая запись содержит большое количество относящихся к ней метаданных, но она также содержит список всех особых характеристик пациента, которые приходится учитывать при использовании информации из записи (эти характеристики могут быть представлены такими записями, как "отказ от курения", "диабет", которые разъясняют содержимое записи). Список особых характеристик пациента представлен в объекте записи в виде списка List<CaseManagementIssue>. Для получения этого списка таблица casemgmt_note должна объединяться с таблицей casemgmt_issue_note (которая выступает в роли таблицы соединения) и, наконец, с таблицей casemgmt_issue.

Если вы захотите написать специфический запрос для Hibernate, что требуется в описанной выше ситуации, вам не придется использовать стандартный язык SQL - вместо него нужно использовать

HQL (язык запросов Hibernate - Hibernate Query Language), который впоследствии будет преобразован в SQL (путем вставки внутренних имен столбцов для всех полей выборки) перед вставкой параметров и отправкой запроса серверу базы данных. В этом специфическом случае для написания запроса были использованы стандартные объединения без объединения столбцов и это означает, что тогда, когда запрос в конечном счете был преобразован в представление SQL, он был настолько длинным, что не было понятно, манипуляции с какими данными осуществляются. В дополнение к этому практически во всех случаях этот запрос не создает достаточно большой таблицы, процесс создания которой можно было бы заметить. Для большинства пользователей этот запрос выполняется достаточно быстро и поэтому он незаметен. Однако, эффективность этого запроса невероятно низка.

Давайте на секунду сделаем шаг назад. В тот момент, когда вы выполняете объединение двух таблиц, серверу приходится создавать временную таблицу в памяти. В большинстве случаев применения стандартных типов объединений количество строк результирующей таблицы равняется количеству строк первой таблицы, умноженному на количество строк второй таблицы. Таким образом, в том случае, если ваша таблица содержит 500,000 строк и вы объединяете ее с таблицей, содержащей 10,000,000 строк, вы создаете временную таблицу, состоящую из 5×10^{12} строк в памяти, после чего из нее осуществляется выборка данных и занятая память освобождается.

В одном из экстремальных случаев, с которым мы столкнулись, в результате объединения трех таблиц была создана временная таблица размером около 7×10^{12} строк, из которой в конечном счете было извлечено около 1000 строк. Эта операция заняла 5 минут и таблица `casemgmt_note` была заблокирована в течение всего времени ее выполнения.

В конечном счете проблема была решена путем использования заранее подготовленных объявлений, которые ограничивали область выборки в первой таблице до момента ее объединения с двумя другими таблицами. Новый, гораздо более эффективный запрос снизил количество строк для выборки до приемлемого значения 300,000 и значительно увеличил производительность операции получения данных записей (время, необходимое для осуществления такой же выборки сократилось до 0.1 секунды).

Мораль этой истории достаточно проста: хотя Hibernate и выполняет свою работу достаточно хорошо, но пока операция объединения таблиц не достаточно точно описана и управляема (либо с помощью файла с расширением `.hbm.xml`, либо с помощью аннотации объединения в рамках класса модели JPA), она может очень быстро выйти из-под контроля. Работа с объектами вместо SQL-запросов требует от вас передачи инициативы по реализации запроса библиотеке для доступа к базе данных и в реальности позволяет вам контролировать исключительно описание операции. Пока вы не начнете тщательно описывать операции, эта библиотека может работать некорректно в экстремальных условиях. Более того, в том случае, если вы разработчик баз данных, обладающий знаниями в области языка запросов SQL, эти знания не окажутся особо ценными при проектировании класса с поддержкой JPA, который лишает вас некоторого контроля над запросами, которым вы могли бы обладать в случае самостоятельной разработки SQL-запросов. В конечном счете можно сделать вывод о том, что для работы необходимы хорошие знания как в области языка SQL, так и в области аннотаций JPA, а также понимание того, как эти аннотации влияют на результирующие запросы.

16.5. Права доступа

Проект CAISI (проект системы клиентского доступа к интегрированным службам и информации - Client Access to Integrated Services and Information) изначально был отдельным продуктом, созданным в результате форка системы OSCAR и предназначенным для управления заведениями для бездомных в Торонто. В конце концов было принято решение о переносе наработок проекта CAISI в форму кода в основную ветку исходного кода. Оригинальный проект CAISI мог больше не раз-

виваться, но при этом из его состава была получена очень важная часть системы OSCAR: модель прав доступа.

Модель прав доступа системы OSCAR предельно мощна и может использоваться для создания такого большого количества ролей и наборов прав доступа, какое только возможно. Провайдеры (*providers*) принадлежат программам (*programs*) (в качестве обслуживающего персонала (*staff*)), в рамках которых они выступают в определенной роли (*role*). Каждая программа находится в учреждении (*facility*). Каждая роль имеет описание (например, "доктор", "медицинская сестра", "социальный работник", и.т.д.) и набор соответствующих глобальных прав доступа. Права доступа записываются в формате, облегчающем их понимание: "read nurse notes" ("читать записи медицинских сестер") может описывать права доступа, которые может иметь роль доктора, при этом роль медицинской сестры может не иметь права доступа "read doctor notes" ("читать записи доктора").

Этот формат может быть простым для понимания, но при более подробном рассмотрении оказывается, что он требует немного сложной работы для проверки таких типов прав доступа. Название роли, в которой выступает рассматриваемый провайдер, проверяется по списку прав доступа путем поиска совпадения с действием, попытка выполнения которого предпринимается. Например, попытка провайдера прочитать записи доктора приведет к проверке права доступа "read doctor notes" для каждой из записей, созданных доктором.

Другой проблемой является преимущественное использование английского языка для описания прав доступа. Любой пользователь системы OSCAR, применяющий отличающийся от английского язык все еще должен описывать права доступа в формате, подобном следующему "read [роль] notes" с использованием таких английских слов, как "read", "write", "notes" и других.

Модель прав доступа системы CAISI является важной частью системы OSCAR, но она не является единственной доступной моделью. Перед тем, как была реализована система CAISI, разрабатывалась другая модель на основе ролей (но не на основе программ), которая и на сегодняшний день используется во многих частях системы.

В этой модели провайдеры ставятся в соответствие одной или нескольким ролям (например, "доктор", "медицинская сестра", "администратор" и другим). Они могут выступать в таком количестве ролей, в каком это необходимо - соответствующие ролям разрешения размещаются друг над другом в стеке. Эти права доступа главным образом используются для запрета доступа к частям системы в отличие от прав доступа системы CAISI, которые запрещают доступ к определенным наборам данных из карты пациента. Например, пользователю необходимо иметь право доступа "_admin" "read" в рамках используемой роли для того, чтобы иметь возможность доступа к панели администрирования. Однако, право доступа "read" лишит пользователей возможности выполнения административных задач. Для выполнения этих задач ему потребуется также право доступа "write".

Обе эти системы предназначены для выполнения практически одной и той же задачи; из-за более поздней интеграции кода системы CAISI в ходе жизненного цикла проекта рассматриваемые системы существуют параллельно. Данные системы не всегда успешно соседствуют друг с другом, поэтому в реальности гораздо проще сфокусироваться на использовании одной из них для повседневной эксплуатации системы OSCAR. В общем случае вы можете оценить возраст кода системы OSCAR, зная о том, какие модели прав доступа предшествовали другим моделям: *модель на основе типов провайдеров (Provider Type)*, *модель на основе ролей провайдеров (Provider Roles)*, *модель на основе программ и провайдеров системы CAISI (CAISI Programs/Roles)*.

Старейшим типом модели прав доступа является модель на основе типов провайдеров ("Provider Type"), которая настолько устарела, что сегодня не используется в большинстве частей системы и фактически использует исключительно роль "доктор" в ходе создания нового провайдера, так как использование любой другой роли (такой, как "регистратор") приводит к проблемам во всей сис-

теме. Управление правами доступа при использовании модели на основе ролей провайдера (Provider Roles) вместо описанной выше может осуществляться проще и точнее.

16.6. Интегратор

Компонент с названием "интегратор" системы OSCAR является отдельным независимым от OSCAR веб-приложением, которое используется экземплярами системы для обмена информацией о пациентах, программах и провайдерах по защищенному каналу. Дополнительно он может быть установлен в качестве компонента такого окружения, как LHN (локальная сеть учреждения здравоохранения - Local Health Network) или клиники. В простейшем случае интегратор может быть описан, как система временного хранения данных.

Представим следующий вариант, аргументирующий необходимость использования интегратора: в составе клиники X существует ЛОР-отделение (занимающееся лечением заболеваний ушей, носа и горла), а также эндокринологическое отделение. В том случае, если ЛОР-врач отправляет пациента к эндокринологу на этаж выше, ему может потребоваться передать вместе с пациентом историю болезни и дополнительные записи. В данном случае неудобно использовать бумагу, тем более бумаг может оказаться больше, чем необходимо - возможно, пациенту после однократного визита больше не понадобится посещать эндокринолога. При использовании интегратора доступ к данным пациента может быть осуществлен с использованием системы электронных медицинских записей эндокринолога, причем доступ к содержимому карты пациента может быть закрыт после его визита.

Более экстремальный пример: в том случае, если человек доставляется в клинику в бессознательном состоянии и система электронных медицинских записей не может найти ничего, кроме его карты медицинского страхования, в случае соединения систем его домашней клиники и госпиталя с помощью интегратора, появляется возможность получения записей из истории болезни данного человека, на основании которых можно быстро выяснить то, что ему был прописан антикоагулянт под названием "варфарин". В конечном счете, функция извлечения информации, похожая на описанную выше, является возможностью, которая может быть реализована такими системами электронных медицинских записей, как OSCAR совместно с интегратором.

Технические подробности

Интегратор доступен исключительно в форме исходного кода, что требует от пользователя действий, направленных на его получение и ручную сборку. Как и в случае системы OSCAR он выполняется в окружении, создаваемом путем стандартной установки Tomcat с MySQL.

При доступе к URL, используемому интегратором, последний не выводит какой-либо полезной информации. Этот компонент является практически в полной мере веб-сервисом; система OSCAR взаимодействует с URL интегратора посредством отправки POST- и GET-запросов.

Являясь независимо разрабатываемым проектом (изначально частью проекта CAISI), интегратор довольно строго следует шаблону проектирования MVC. Разработчики оригинальных версий проекта проделали великолепную работу, изначально очень четко установив связи между моделями, представлениями и контроллерами. Наиболее новым реализованным типом уровня доступа к базе данных, о котором я упоминала ранее, является стандартный уровень, использующий технологию JPA и являющийся при этом единственным подобным уровнем проекта. (В качестве интересного примечания следует упомянуть о том, что из-за использования аннотации JPA для всех классов моделей в рамках всего проекта сценарий для работы с SQL создается во время сборки и может использоваться для инициализации структуры базы данных; следовательно, в комплекте поставки интегратора нет отдельного сценария для работы с SQL.)

Взаимодействие с интегратором происходит с помощью отправки запросов веб-сервиса, описанных в файлах WSDL XML, которые доступны на сервере. Клиент может отправить запрос интегратору для выяснения того, какие типы функций доступны и использовать их. Фактически это означает, что интегратор совместим с любым типом системы электронных медицинских записей для которой кто-либо захочет разработать код клиента; формат данных является достаточно общим, поэтому он хорошо совместим с локальными типами данных.

При этом в случае системы OSCAR клиентская библиотека встроена и включена в основное дерево исходного кода для упрощения использования. Обновление этой библиотеки требуется только тогда, когда становятся доступны новые функции на стороне интегратора. Исправление ошибок интегратора не требует обновления этого файла.

16.6. Интегратор

Архитектура

Данные для работы интегратора передаются со стороны всех соединенных с ним систем электронных медицинских записей в установленное время и после их доставки другая система электронных медицинских записей может запросить эти данные. При этом никакие из данных не хранятся интегратором на постоянной основе - его база данных может быть очищена и заполнена заново данными от клиентов.

Набор отправляемых данных настраивается индивидуально на стороне каждого из экземпляров системы OSCAR, соединенного с определенным интегратором и, за исключением случаев, когда база пациентов в полном объеме должна быть передана серверу интегратора, ему передаются только те записи пациентов, которые были просмотрены с момента последней отправки данных. Этот процесс не является точной копией процесса наложения патча, но очень похож на него.

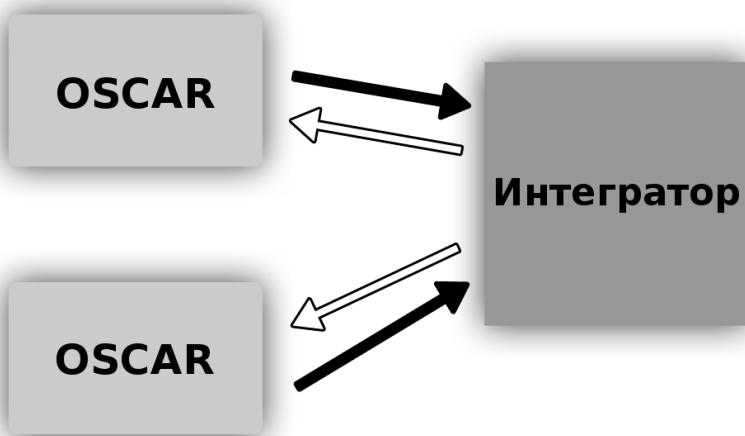


Рисунок 16.1: Обмен данными между системами OSCAR и интегратором

Позвольте мне объяснить принцип работы интегратора более подробно с помощью примера: в удаленной клинике хотят посмотреть карту пациента из другой клиники. При желании персонала этой клиники получить доступ к записи пациента, следует первую очередь подключить эти клиники к одному и тому же интегратору. Регистратор может осуществить поиск удаленного пациента с помощью интегратора (по имени и в случае необходимости по дате рождения или полу) и найти запись необходимого пациента, хранимую на сервере. Он инициирует копирование ограниченного набора демографической информации пациента, после чего подтвердит намерения пациента, получив его согласие на извлечение данных путем заполнения формы согласия. После этого сервер интегратора передаст всю информацию, известную интегратору о пациенте - записи, сведения о прописанных препаратах, аллергических реакциях, прививках, дополнительные документы, и.т.д. Эти данные кэшируются локально, поэтому локальной копии системы OSCAR не придется от-

правлять интегратору запрос каждый раз, когда потребуется обратиться к этим данным, но следует учесть и то, что срок хранения локального кэша ограничивается одним часом.

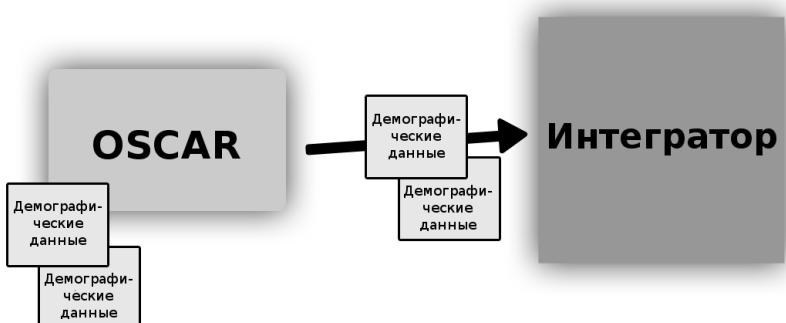


Рисунок 16.2: Демографическая информация и соответствующие данные отправляются интегратору в процессе передачи данных из домашней клиники. Хранящаяся на интеграторе запись может быть не полным представлением записи из домашней клиники, так как система OSCAR позволяет выбрать вариант отправки данных пациента не в полном объеме.

После начальной настройки записи пациента, выполненной путем копирования его демографических данных в локальную систему OSCAR, запись пациента начинает работать точно так же, как и любая другая запись в рамках системы. Все данные с удаленной системы, принятые от интегратора получают соответствующую метку (также вместе с данными сохраняется информация о клинике, из которой они получены), но они всего лишь временно кэшируются в рамках локальной системы OSCAR. Любые сохраняемые локальные данные записываются абсолютно также, как и любые другие данные пациента в записи пациента, после чего отправляются интегратору, но при этом они не хранятся на удаленной машине на постоянной основе.

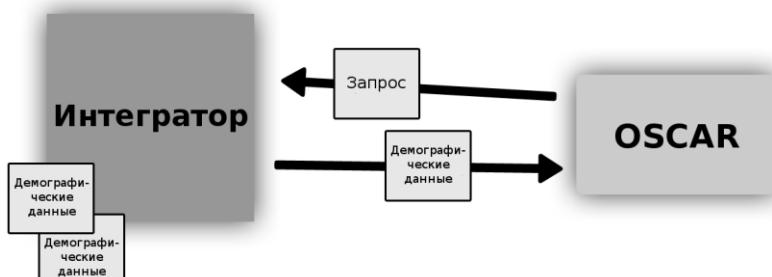


Рисунок 16.3: Удаленная система OSCAR запрашивает данные у интегратора, указывая на определенную запись пациента. Сервер интегратора отправляет исключительно демографическую информацию, которая хранится на постоянной основе удаленной системой OSCAR.

Этот процесс имеет очень важное значение, особенно для получения согласия пациента и понимания воздействия описанных факторов на архитектуру интегратора. Представим, что пациент посетил врача из удаленной клиники и согласился на предоставление доступа к своей записи, но только на некоторое время. После визита пациент может аннулировать соглашение о возможности открытия персоналом этой клиники его записи и в следующий раз при открытии карты пациента из этой клиники данные не будут доступны (за исключением тех данных, которые были сохранены локальной системой). В конечном счете этот механизм позволяет пациенту непосредственно контролировать то, как и когда его запись может быть просмотрена аналогично посещению клиники с бумажной копией карты пациента. Персонал клиники сможет ознакомиться с картой только при общении с вами, но вы заберете ее домой тогда, когда покинете клинику.

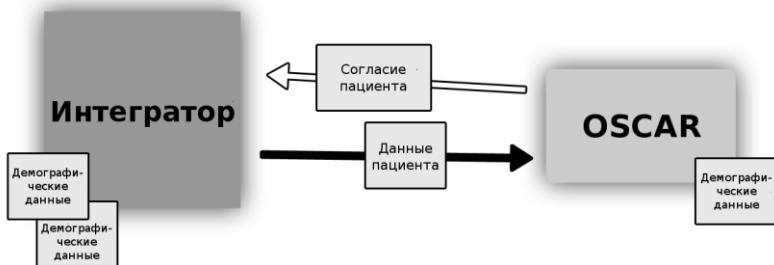


Рисунок 16.4: Персонал удаленной клиники может просматривать содержимое карты пациента, запрашивая данные; в случае согласия пациента данные передаются. Данные никогда не хранятся на постоянной основе удаленной системой OSCAR.

Другой очень важной для медиков возможностью является возможность принятия решения о том, какие данные они хотели бы передавать другим присоединенным клиникам посредством своего сервера интегратора. Персонал клиники может выбрать передачу всех данных из демографической записи или только частей этой записи, например, записей без документов, информации об аллергических реакциях без предписаний, и.т.д. В конечном счете решение о том, какими типами данных было бы удобно делиться друг с другом принимается группой медиков, настраивающих сервер интегратора.

Как я упоминала ранее, интегратор является исключительно системой временного хранения данных и с помощью него данные никогда не сохраняются на постоянной основе. Это еще одно очень важное решение, которое было принято в ходе разработки; оно позволяет клиникам очень просто отказаться от передачи всех данных с помощью интегратора и, фактически в случае необходимости может быть очищена вся база данных интегратора. В случае очистки базы данных пользователи клиентских систем не заметят этого, так как данные будут аккуратно реконструированы на основе исходных данных, находящихся на различных соединенных с интегратором клиентских системах. Эта возможность влечет за собой требование, согласно которому предоставляющая данные система OSCAR должна доверять получающему данные интегратору, предоставляя возможность очистки базы данных по первому требованию, следовательно, лучшим решением является развертывание интегратора группой медицинских работников из такой зарегистрированной согласно требованиям законодательства организации, как Family Health Organization или Family Health Team; в этом случае сервер интегратора будет эксплуатироваться одной из клиник, в которой работают эти специалисты.

Формат данных

Клиентские библиотеки интегратора созданы с помощью программного компонента `wsd2java`, который формирует набор классов, представляющих соответствующие типы данных, используемые веб-сервисом в ходе взаимодействия с клиентами. Среди них присутствуют классы для каждого из типов данных наряду с классами, представляющими ключи для каждого из этих типов данных.

Описание способа сборки клиентской библиотеки интегратора выходит за границы данной главы. Единственной важной вещью в данной связи является информация о том, что сразу же после сборки библиотеки она должна быть добавлена в комплект поставки системы OSCAR ко всем остальным JAR-файлам. Этот JAR-файл содержит все необходимое для установления соединения с интегратором и доступа ко всем типам данных, которые сервер интегратора будет передавать системе OSCAR, такими, как `CachedDemographic`, `CachedDemographicNote` и `CachedProvider` наряду с многими другими. В дополнение к типам данных, которые передаются, существуют "WS"-классы, которые используются в первую очередь для получения таких наборов данных, как наиболее часто используемый класс `DemographicWs`.

Работа с данными интегратора иногда может оказаться немного усложненной. Система OSCAR не имеет каких-либо жестко интегрированных механизмов для работы с типом данных, который обычно используется при получении определенного типа информации о пациенте (например, за-

писей из медицинской карты) в момент, когда клиент интегратора отправляет запрос для получения данных от сервера. После этого данные вручную преобразуются в локальный класс, представляющий эти данные (в случае записей это класс `CaseManagementNote`). В рамках класса типа данных устанавливается логический флаг, указывающий на то, что этот класс содержит данные от удаленной системы и влияющий на то, как данные будут выводиться на экран пользователя. С другой стороны, класс `CaisIntegratorUpdateTask` обрабатывает выборку локальных данных системой OSCAR, преобразование их в формат данных интегратора и последующую отправку этих данных серверу интегратора.

Эта архитектура может и не являться такой эффективной и прозрачной, как могла бы, но она позволяет устаревшим в большей степени частям системы стать "совместимыми" с доставляемыми интегратором данными без значительных модификаций. В дополнение к этому поддержание представления таким простым, как это возможно путем осуществления обращений только к одному типу класса улучшает читаемость JSP-файла и упрощает процесс отладки в случае обнаружения ошибки.

16.7. Выученные уроки

Как вы, возможно, представляете, система OSCAR имеет свой набор недостатков, присущих ее архитектуре. Однако, она предоставляет завершенный набор возможностей, с которым у большинства пользователей не возникает никаких проблем. В этом и заключается главная цель проекта: предоставить качественное решение, функционирующее в большинстве ситуаций.

Я не могу говорить от лица всего сообщества разработчиков системы OSCAR, поэтому в данном разделе будет отражена лишь моя субъективная точка зрения. Мне кажется, что существуют некоторые важные темы, не относящиеся к аспектам архитектуры проекта.

Во-первых, очевидно, что недостаточный контроль над исходным кодом в прошлом привел к тому, что архитектура системы стала достаточно неупорядоченной в некоторых местах, особенно в тех областях, где контроллеры и представления смешивались друг с другом. Путь, по которому проект развивался в прошлом, не позволил предотвратить подобные последствия, но с того времени процесс разработки претерпел значительные изменения и, надеюсь, что проект не столкнется с подобной проблемой снова.

Еще следует упомянуть о том, что из-за достаточного возраста проекта становится сложно обновить (или даже изменить) библиотеки без причинения значительных неудобств остальной кодовой базе. Кстати, это именно ситуация, которая и произошла. Мне обычно сложно выяснить, что важно, а что нет при исследовании директории библиотек. В дополнение к этому следует упомянуть о том, что иногда при значительных обновлениях библиотек они нарушают обратную совместимость (изменение названий пакетов является стандартной причиной). Обычно в комплекте поставки системы OSCAR присутствует несколько библиотек, выполняющих одну и ту же задачу - это результат недостаточного контроля над исходным кодом, а также того факта, что не было создано документации со списком библиотек и описанием того, какая библиотека требуется для какого из компонентов.

Дополнительно следует отметить то, что проект OSCAR не достаточно гибок при добавлении новых функций в существующие подсистемы. Например, в том случае, когда вы хотите добавить новое поле в электронную карту пациента, вам придется создать новую JSP-страницу и новый сервлет, модифицировать шаблон электронной карты (в нескольких местах) и модифицировать конфигурационный файл приложения для возможности загрузки вашего сервлета.

Кроме того, ввиду отсутствия документации иногда практически невозможно выяснить то, как работает часть системы - человек, разработавший оригинальный код, может уже не участвовать в проекте и обычно единственным доступным инструментом, позволяющим вам выяснить это, явля-

ется отладчик. В случае рассмотрения проекта такого возраста ценой этого обстоятельства являются утраченные возможности новых потенциальных участников, которые могли бы начать работу в рамках проекта. Однако, благодаря совместным усилиям участников проекта и схожим факторам, сообщество продолжает работу.

Наконец, система OSCAR является репозиторием медицинской информации и ее безопасность значительно снижается из-за включения в комплект поставки класса `DBHandler` (описанного в предыдущем разделе). Лично я считаю, что принимающие параметры запросы к базе данных в свободной форме ни в коем случае не должны быть разрешены в системе электронных медицинских записей, так как с помощью них можно достаточно просто производить атаки на основе SQL-инъекций. Хотя запрет на разработку нового, использующего данный класс кода и является правильным решением, приоритетная задача команды разработчиков должна заключаться в удалении всех вариантов использования данного класса.

Все эти слова могут звучать как жесткая критика проекта. В прошлом все описанные проблемы были заметны и, как я говорила, они сдерживали рост сообщества из-за высокого барьера входления в проект. Но ситуация меняется, поэтому в будущем эти проблемы не будут настолько большиими препятствием на пути развития.

Оглядываясь назад и рассматривая историю развития проекта (особенно в течение выхода нескольких последних версий), мы можем лучшим образом спроектировать приложение. Система все так же должна будет предоставлять базовый набор функций (установленный правительством Онтарио для сертификации приложения как системы электронных медицинских записей), поэтому эти функции должны быть реализованы по умолчанию. Но в том случае, если архитектура системы OSCAR будет изменяться сегодня, она должна стать по-настоящему модульной и позволять рассматривать модули как плагины; если вам не нравится модуль электронной формы, вы получите возможность создать свою собственную реализацию (или даже полностью отличный модуль). У системы должна появиться возможность взаимодействия с большим количеством систем (или большее количество систем должно иметь возможность взаимодействия с ней), включая медицинское оборудование, которое все чаще используется в индустрии, такое, как оборудование для проверки остроты зрения. Это утверждение также обозначает, что должна быть предоставлена возможность достаточно простой адаптации системы OSCAR к требованиям, предъявляемым к системам хранения медицинских данных региональными и федеральными правительствами всех стран мира. Так как каждый регион имеет отличный от других набор законов и требований, это архитектурное решение должно быть ключевым для уверенности в том, что система OSCAR разрабатывается для нужд пользователей со всего мира.

Я также верю в то, что вопрос безопасности должен быть наиболее важным из всех. Система электронных медицинских записей безопасна ровно настолько, насколько безопасен ее наименее защищенный компонент, поэтому особое внимание должно быть уделено вопросу абстрагирования приложения от метода доступа к данным настолько, насколько это возможно таким образом, чтобы оно хранило и получало данные, работая в безопасном окружении и используя основной уровень API доступа к данным, который прошел аудит сторонних лиц и считается подходящим для хранения медицинской информации. Другие системы электронных медицинских записей могут скрывать подробности реализации и использовать закрытый проприетарный код в качестве меры безопасности (которая на самом деле не является таковой), но так как система OSCAR распространяется в форме открытого исходного кода, она должна возглавлять список наиболее тщательно защищающих данные систем.

Я твердо верю в будущее проекта OSCAR. У нас есть сотни пользователей, о которых мы знаем (а также многие сотни пользователей, о которых нам не известно) и мы получаем важные отчеты о работе приложения от медицинских работников, которые взаимодействуют с нашим проектом ежедневно. Через разработку новых процессов и добавление новых возможностей мы надеемся расширить базу установок и начать поддерживать пользователей из всех других регионов. Нашим

намерением движет уверенность в том, что мы предоставляем что-либо, улучшающее жизнь использующих систему OSCAR медицинских работников, а также жизни их пациентов, разрабатывая лучшие инструменты для упрощения работы в сфере здравоохранения.

17. Processing.js

Глава 17 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Изначально разработанный Ben Fry и Casey Reas, язык программирования Processing начал свое развитие в виде языка программирования с открытым исходным кодом (на основе Java), созданного для помощи участникам сообществ дизайнеров и других творческих личностей в изучении программирования в визуальном контексте. Предоставляя значительно упрощенную модель обработки двумерной и трехмерной графики по сравнению с большинством языков программирования, он был быстро адаптирован для выполнения широкого круга задач, начиная с обучения программированию путем разработки простых визуализаций и заканчивая созданием многостенных художественных инсталляций, а также получил возможность выполнения широкого круга задач, от простого считывания последовательности строк до возможности реализации приложения, фактически являющегося интегрированной средой разработки для программирования и управления популярными платами, предназначенными для создания прототипов на основе открытого аппаратного обеспечения под названием "Arduino". Продолжая набирать популярность, Processing твердо занял свое место легко изучаемого и широко используемого языка программирования для создания любых типов визуализаций, а также выполнения большого количества других задач.

Простейшая программа на языке программирования Processing, называемая "скетч" ("sketch"), состоит из двух функций: `setup` и `draw`. Первая функция является основной точкой входа и может содержать любое количество инструкций инициализации. После завершения выполнения функции `setup` программы на языке Processing могут использовать один из следующих вариантов продолжения работы: 1) вызвать функцию `draw` и запланировать следующий вызов функции `draw` по истечении фиксированного временного интервала после завершения работы функции; 2) вызвать функцию `draw` и ждать событий ввода от пользователя. По умолчанию язык программирования Processing использует первый вариант; вызов функции `noLoop` приводит к использованию второго варианта. Это обстоятельство позволяет использовать два режима представления скетчей, а именно: режим работы в графическом окружении с фиксированной частотой кадров и интерактивный режим, в котором графическое окружение обновляется в ходе обработки событий. В обоих случаях пользовательские события отслеживаются и могут быть обработаны либо с помощью их собственных обработчиков, либо путем установки постоянных значений глобальных переменных напрямую в функции `draw` для определенных событий.

Processing.js является родственным проектом для проекта Processing, спроектированным с целью его переноса в веб-пространство без необходимости использования виртуальной машины Java или плагинов. Его развитие началось с попытки установления John Resig того, может ли язык Processing быть портирован для использования в веб-пространстве путем работы с новым на тот момент элементом `<canvas>` из состава HTML5 в роли графического контекста при использовании прототипа библиотеки, представленного в 2008 году. Разработанная с мыслью о том, что "код должен просто работать", библиотека Processing.js совершенствовалась в течение долгих лет с целью предоставления возможности создания визуализаций данных, цифровых произведений искусства, интерактивных анимаций, обучающих графиков, видеоигр и других работ с использованием веб-стандартов без каких-либо плагинов. Вы можете разрабатывать код на языке программирования Processing, используя либо Processing IDE, либо ваш любимый текстовый редактор, интегрировать его в веб-страницу, используя элемент `<canvas>`, после чего библиотека Processing.js выполнит остальные шаги, выводя необходимые графические данные с использованием элемента `<canvas>` и позволяя пользователям взаимодействовать с графическими данными таким же образом, каким они могли бы взаимодействовать с обычным обособленным Processing-приложением.

17.1. Как это работает?

Библиотека Processing.js является немного необычной для проекта с открытым исходным кодом, так как ее кодовая база содержится в единственном файле с именем `processing.js`, который содержит код для реализации функций языка Processing, представленный единственным объектом, реализующим функции всей библиотеки. При обсуждении метода структурирования кода следует упомянуть о том, что мы постоянно перемещаем элементы этого объекта, пытаясь немного улучшить код в каждом релизе. Архитектура библиотеки достаточно проста, а ее функция может быть описана одним предложением; она преобразует исходный код на языке Processing в корректный исходный код на языке JavaScript, причем каждый вызов функции API языка Processing ставится в соответствие подходящей функции объекта преобразования элементов языка Processing в JavaScript, что в итоге приводит к выполнению с элементом `<canvas>` тех же действий, которые были бы выполнены при использовании языка Processing с канвой апплета Java.

Для повышения скорости работы приложений мы используем два отдельных пути исполнения кода для работы с 2D- и 3D-функциями, при этом при загрузке скетча используется либо первый, либо второй путь для выбора необходимых оберток функций, из чего можно сделать вывод о том, что мы избегаем увеличения затрат ресурсов экземплярами выполняющихся приложений. Однако, при разговоре о структурах данных и направлении выполнения кода, следует учитывать тот факт, что знания в области языка JavaScript подразумевают то, что вы сможете прочитать код файла `processing.js`, возможно за исключением системы разбора синтаксических структур.

Унификация кода на языках Java и JavaScript

Преобразование кода на языке Processing в код на языке JavaScript обозначает, что вы можете просто сообщить браузеру о том, что нужно выполнить полученный в результате преобразования код и в том случае, если вы выполнили преобразование корректно, он просто заработает. Но уверенность в том, что преобразование выполнено и время от времени также выполняется корректно, требует приложения некоторых усилий. Синтаксис языка программирования Processing основан на синтаксисе языка программирования Java и это значит, что библиотеке Processing.js приходится в общем случае преобразовывать исходный код на языке Java в исходный код на языке JavaScript. Изначально эта процедура выполнялась путем рассмотрения исходного кода на языке Java в форме строки и итерационной замены специфичных для Java подстрок на их аналоги из JavaScript. (Интересующиеся ранней версией системы разбора кода читатели могут найти ее код [здесь](#), причем следует рассматривать фрагмент кода со строки 37 до строки 266). Для небольшого синтаксического набора это решение было приемлемым, но с течением времени и нарастанием сложности оно начало давать сбои. Впоследствии система разбора кода была полностью переписана для добавления возможности построения абстрактного синтаксического дерева (Abstract Syntax Tree - AST) вместо разбора строки, путем изначального разбиения исходного кода на языке Java на функциональные блоки и последующего сопоставления каждого из таких блоков с соответствующими синтаксическими конструкциями языка JavaScript. В результате, хотя и произошло снижение читаемости кода библиотеки Processing.js, в ее составе появился транскомпилятор, преобразующий код на языке Java в код на языке JavaScript в процессе работы приложения. (Читатели могут внимательно изучить [этот код](#) вплоть до строки 19217.)

Ниже приведен код скетча на языке Processing:

```
void setup() {
    size(200, 200);
    noCursor();
    noStroke();
    smooth(); }

void draw() {
```

```

fill(255,10);
rect(-1,-1,width+1,height+1);
float f = frameCount*PI/frameRate;
float d = 10+abs(60*sin(f));
fill(0,100,0,50);
ellipse(mouseX, mouseY, d,d); }

```

И этот же код после преобразования с помощью библиотеки Processing.js:

```

function($p) {
    function setup() {
        $p.size(200, 200);
        $p.noCursor();
        $p.noStroke();
        $p.smooth(); }
    $p.setup = setup;

    function draw() {
        $p.fill(255, 10);
        $p.rect(-1, -1, $p.width + 1, $p.height + 1);
        var f = $p.frameCount * $p.PI / $p._frameRate;
        var d = 10 + $p.abs(60 * $p.sin(f));
        $p.fill(0, 100, 0, 50);
        $p.ellipse($p.mouseX, $p.mouseY, d, d); }
    $p.draw = draw; }

```

Все это звучит здорово, но существует несколько проблем, которые мешают преобразованию синтаксических конструкций языка Java в синтаксические конструкции языка JavaScript:

1. Программы на языке Java являются изолированными объектами. Программы на языке JavaScript делят данные с веб-страницей.
2. Язык Java использует строгую типизацию. JavaScript не использует ее.
3. Java является основанным на классах и их экземплярах объектно-ориентированным языком программирования. JavaScript не является таковым.
4. Язык Java использует разделенные переменные и методы. JavaScript не проводит такого разделения.
5. Язык Java позволяет производить перегрузку методов. JavaScript не предоставляет такой возможности.
6. Язык Java позволяет производить импорт скомпилированного кода. В рамках JavaScript не вводится даже такого понятия.

Решение этих проблем было компромиссом между тем, что нужно пользователям и тем, что мы можем сделать с помощью веб-технологий. В последующих разделах мы обсудим каждую из этих проблем более подробно.

17.2. Значительные различия

Программы на языке Java работают в своих собственных потоках; программы на языке JavaScript могут заблокировать ваш браузер.

Программы на языке Java являются изолированными объектами, выполняющимися в своих собственных потоках, находящихся в большом наборе выполняемых приложений в вашей системе. Программы на языке JavaScript, напротив, выполняются в браузере и соперничают друг с другом образом, не свойственным обычными приложениями для настольных компьютеров. В тот момент, когда программа на языке Java загружает файл, она ожидает окончания загрузки ресурса, после чего выполнение программы в обычном режиме продолжается. В том случае, когда программа является изолированным объектом, этот подход приемлем. Операционная система может отвечать на действия пользователя, так как она ответственна за планирование выполнения программных потоков и даже в том случае, когда программе требуется час для загрузки всех необходимых ей данных, вы также можете использовать свой компьютер. В случае веб-страницы процесс работы приложения значительно отличается. Если ваша "программа" на языке JavaScript ожидает загрузки ресурса, она заблокирует свой процесс до тех пор, пока ресурс не станет доступен. В том случае,

если вы используете браузер, в котором для каждой вкладки создается отдельный процесс, ваша вкладка будет заблокирована, но браузер все равно можно будет использовать. Если вы используете браузер, не создающий таких процессов, он может перестать отвечать на запросы пользователя. Таким образом, независимо от назначения процесса, страница, на которой выполняется сценарий не сможет использоваться до момента загрузки ресурса, причем возможно, что ваш интерпретатор JavaScript полностью заблокирует браузер.

Такое поведение неприемлемо для современных веб-приложений, в которых ресурсы передаются асинхронно и страница должна работать в нормальном режиме в процессе фоновой загрузки ресурсов. Хотя это поведение и свойственно для традиционных веб-страниц, для веб-приложений оно становится реальной головной болью: как вы заставите код на языке JavaScript бездействовать в течение заданного промежутка времени в ожидании загрузки ресурса при условии того, что в рамках языка JavaScript не существует явного механизма для перевода приложения в режим ожидания? Хотя в рамках языка JavaScript также нет явного механизма для работы с потоками, он реализует модель событий и объект XMLHttpRequest, предназначенный для запроса произвольных (представленных не только в форматах XML или HTML) данных с использованием произвольных строк URL. Этот объект поддерживает несколько различных событий состояния и мы можем использовать его асинхронно для получения данных, причем в процессе браузер будет отвечать на запросы пользователя. Этот объект отлично подходит для программ, в которых осуществляется управление исходным кодом: вы просто останавливаете программу после осуществления запроса данных и возобновляете выполнение программы в момент, когда данные становятся доступны. Однако, это практически невозможно для кода, разработанного в соответствии с идеей синхронной загрузки ресурсов. Вставка "периодов ожидания" в программы, предназначенные для работы в режиме фиксированной частоты кадров не является приемлемой, поэтому нам придется прибегнуть к альтернативным подходам.

Все же в некоторых случаях мы решили применить операции синхронного ожидания. Например, при загрузке файла со строками используется синхронная версия объекта XMLHttpRequest, которая заблокирует выполнение сценариев страницы до тех пор, пока данные не станут доступны. В других случаях нам приходилось проявлять сообразительность. Для загрузки изображений, например, использовался встроенный механизм загрузки изображений браузера; мы создавали новый объект `Image` с помощью JavaScript, устанавливали строку URL изображения в качестве значения его атрибута `src`, после чего браузер выполнял всю остальную работу, уведомляя нас о том, что изображение доступно с помощью события `onload`. Этот механизм даже не основывается на объекте XMLHttpRequest, он просто эксплуатирует возможности браузера.

Для упрощения работы в том случае, если вам заранее известно о том, какие изображения должны быть загружены, мы добавили поддержку директив предварительной загрузки и, таким образом, выполнение скетча не будет начато до тех пор, пока процесс предварительной загрузки изображений не будет завершен. Пользователь может задать любое количество изображений для предварительной загрузки с помощью блока комментариев в начале скетча; после этого библиотека Processing.js выполнит предварительную загрузку изображений. Событие `onload` для каждого из изображений сообщит нам о том, что передача данных изображения закончена и оно подготовлено для вывода (изображение просто загружено, но пока не декодировано в массив пикселей), после чего мы можем загрузить данные в соответствующий объект Processing с именем `PImage` с указанием корректных значений (`width` (ширина), `height` (высота), данные пикселей, и.т.д.) и удалить изображение из списка предварительной загрузки. В тот момент, когда список становится пустым, начинается выполнение скетча и используемые в ходе его выполнения изображения могут использоваться без необходимости ожидания их загрузки.

Ниже приведен пример директив предварительной загрузки:

```
/* @pjs preload=".//worldmap.jpg"; */
```

```

PImage img;

void setup() {
  size(640, 480);
  noLoop();
  img = loadImage("worldmap.jpg"); }

void draw() {
  image(img, 0, 0); }

```

Для других случаев нам пришлось разработать более запутанные системы "ожидания доставки ресурсов". Шрифты, в отличие от изображений, не имеют соответствующих встроенных в браузер механизмов загрузки (или, по крайней мере, системы, настолько же функциональной, насколько система загрузки изображений). Хотя загрузка шрифтов и может быть осуществлена с помощью правила CSS `@font-face`, причем она будет осуществляться силами браузера, не существует событий JavaScript для уведомления об окончании загрузки шрифта. Мы наблюдаем медленный процесс добавления в браузеры функций, предназначенных для генерации событий JavaScript, указывающих на завершение процесса загрузки шрифтов, но эти события генерируются "слишком рано", так как браузеру после загрузки шрифта может потребоваться еще от нескольких до нескольких сотен миллисекунд для разбора шрифта перед его использованием на странице. Следовательно, использование этих событий может привести либо к невозможности применения шрифта, либо к применению отличающегося от указанного шрифта в том случае, когда указан шрифт, который должен использоваться в случае ошибки. Вместо использования этих событий мы включили в комплект поставки библиотеки замечательный шрифт формата TrueType, содержащий единственную букву "A" с чрезвычайно малыми метриками и передали браузеру команду для загрузки этого шрифта с помощью правила `@font-face` со строкой URI, содержащей байтовое представление шрифта в форме строки в формате BASE64. Этот шрифт настолько малого размера, что мы можем быть уверены в том, что он будет доступен немедленно. Для любой другой инструкции загрузки шрифта мы сравниваем метрики текста для желаемого и встроенного шрифтов. Для скрытого тэга `<div>` установлен атрибут использования желаемого шрифта, при этом в случае ошибки используется встроенный шрифт. В то время, как текст в рамках этого тэга `<div>` имеет чрезвычайно малые размеры, нам известно, что желаемый шрифт пока не доступен, поэтому мы просто проверяем размеры шрифта через заданные интервалы времени до того момента, как метрики шрифта станут разумными.

Язык Java использует строгую типизацию; JavaScript не использует ее.

В языке Java числовые значения 2 и 2.0 отличаются, поэтому при их использовании в математических операциях будут получены различные результаты. Например, при использовании кода `i = 1/2` значение переменной `i` будет равно 0, так как эти значения рассматриваются как целочисленные, но при этом при использовании кода `i = 1/2.0, i = 1.0/2` и даже `i = 1./2.` значение переменной `i` будет равно 0.5, так как числа рассматриваются как дробные десятичные с ненулевой целой частью и нулевой дробной частью. Даже в том случае, если результирующий тип данных является числовым с плавающей точкой, а в арифметических операциях используются только целочисленные значения, результат также будет целочисленным. Это обстоятельство позволяет вам записывать достаточно сложные математические выражения при использовании языка программирования Java, а, следовательно, и при использовании языка программирования Processing, но они будут генерировать теоретически значительно отличающиеся результаты при переходе к использованию библиотеки Processing.js, так как в рамках языка JavaScript вводится только понятие "чисел". При разговоре о JavaScript следует упомянуть о том, что значения 2 и 2.0 рассматриваются как одно и то же число, что может привести к очень занимательным ошибкам при выполнении скетча с использованием библиотеки Processing.js.

Это может показаться большой проблемой и мы изначально были убеждены именно в этом, но вам не удастся поспорить с реальными отчетами об использовании библиотеки: оказывается, что люди практически никогда не сталкиваются с подобной проблемой при размещении своих скетчей, ис-

пользующих библиотеку Processing.js, в сети. Вместо решения этой проблемы каким-либо замечательным и творческим способом, было предложено удивительно прямолинейное решение; мы не стали решать ее и, принимая архитектурное решение, мы посчитали ненужным пересмотр сложившегося порядка вещей. Если говорить кратко, мы могли добавить таблицу символов строгой типизации и таким образом получить поддержку несуществующих типов в рамках языка JavaScript для переключения функций в зависимости от типа данных, но эта несовместимость не могла быть корректно устранена без сложной работы по поиску неочевидных ошибок, поэтому вместо добавления большого объема кода и замедления процесса выполнения приложений, мы оставили эту особенность работы библиотеки нетронутой. Это хорошо документированная особенность, поэтому "качественный код" не должен пытаться использовать преимущества явных преобразований типов из языка Java. При этом иногда вы можете забыть об этой особенности и результат работы приложения может оказаться весьма интересным.

Java является основанным на классах и их экземплярах объектно-ориентированным языком программирования с четким разделением между пространствами переменных и методов. JavaScript не является таковым.

JavaScript использует прототипы объектов и соответствующую им модель наследования. Это значит, что все объекты представлены в формате пар ключ/значение, где каждый ключ представлен строкой, а значения являются примитивами, массивами, объектами или функциями. При взгляде со стороны наследования следует отметить, что прототипы могут дополнять другие прототипы, но не существует реальной концепции "суперкласса" и "подкласса". Для добавления возможности исполнения "корректного" объектно-ориентированного кода в Java-стиле нам пришлось реализовать классическую модель наследования для языка JavaScript в рамках библиотеки Processing.js без значительного замедления ее работы (нам кажется, что в этом плане мы добились успеха). Нам также пришлось предложить способ предотвращения коллизий между именами переменных и именами функций. Из-за представления объектов в JavaScript в формате ключ/значение, описание переменной с именем `line` с последующим описанием функции, подобным `line(x1, y1, x2, y2)` позволит вам работать с объектом, использующим только то, что было объявлено в последнюю очередь. Сначала JavaScript устанавливает значение `object.line = "some value"`, после чего повторно устанавливает значение переменной `line object.line = function(x1, y1, x2, y2) (...)`, заменяя то значение `line`, которое вы ожидали увидеть.

Создание механизма раздельного управления переменными и методами/функциями значительно замедлило бы библиотеку, поэтому в документации точно так же описано то, что использование функций и переменных с одинаковыми именами является плохой идеей. В том случае, если бы все разрабатывали "корректный" код, это не стало бы большой проблемой, так как обычно переменные и функции называются в зависимости от того, для чего они предназначены или что они делают, но в реальности все не так. Иногда ваш код не будет работать и это произойдет из-за нашего решения, заключающегося в том, что в случае конфликта имен предпочтительным вариантом является отказ от выполнения кода вместо медленной работы в любом случае. Второй причиной, обуславливающей отсутствие реализации разделения между переменными и функциями является то, что такой подход может привести к неработоспособности кода на языке JavaScript, используемого в скетчах системы Processing. Замыкания и цепочка областей видимости языка JavaScript основываются на характере объектов, представленных парами ключ/значение, поэтому вмешательство в процесс их обработки путем разработки собственных методов управления также могло значительно повлиять на производительность, в особенности в областях компиляции при выполнении и компрессии, использующих замыкания функций.

Язык Java позволяет производить перегрузку методов. JavaScript не предоставляет такой возможности.

Одной из наиболее мощных возможностей языка Java является возможность объявления функции, скажем `add(int, int)`, после которой может быть объявлена другая функция с таким же именем,

но отличающимся набором аргументов, т.е., `add(int, int, int)` или с другими типами аргументов, т.е., `add(ComplexNumber, ComplexNumber)`. При вызове функции `add` с двумя или тремя целочисленными аргументами автоматически будет вызвана соответствующая функция, а при вызове функции `add` с аргументами, представленными числами с плавающей точкой или объектами `Car`, будет сгенерирована ошибка. С другой стороны, в языке JavaScript нет поддержки такой возможности. В JavaScript функция является свойством и вы можете разыменовать ее (в этом случае JavaScript передаст вам значение после приведения типов, которое в данном случае будет логическим значением `true`, если свойство указывает на описание функции или `false` в противном случае) или осуществить ее вызов, используя операторы исполнения (которые записываются с помощью скобок, без или с некоторым количеством аргументов в них). Если вы объявите функцию `add(x, y)`, после чего осуществите ее вызов `add(1, 2, 3, 4, 5, 6)`, описанный код будет приемлем для JavaScript. В качестве значения аргумента `x` будет установлено число 1, а в качестве значения `y` - 2, причем все последующие аргументы будут просто проигнорированы. Для того, чтобы сделать механизм перегрузки методов работоспособным, мы заменяем функции с идентичными именами и разным количеством аргументов на пронумерованную функцию, следовательно функция `function(a, b, c)` в исходном коде будет преобразована в функцию `function$3(a, b, c)` в результирующем коде и функция `function(a, b, c, d)` превратится в функцию `function$4(a, b, c, d)`, что позволит использовать корректные пути исполнения кода.

Мы также практически полностью решили проблему перегрузки функций с одинаковым количеством аргументов различных типов, так как типы аргументов могут *дифференцироваться* средствами языка JavaScript. JavaScript может сообщить тип свойств функций при использовании оператора `typeof`, который вернет строку `number`, `string`, `object` или `function` в зависимости от того, что представлено с помощью свойства. Описание `var x = 3` с последующим описанием `x = '6'` приведет к тому, что `typeof x` вернет строку `number` после первого описания и строку `string` после повторного присваивания. Пока функции с одинаковым количеством аргументов отличаются их типами, мы осуществляляем их переименование и выбор в зависимости от результата операции `typeof`. Этот подход не работает в том случае, если функции принимают аргументы типа `object`, поэтому для таких функций мы используем дополнительную проверку с помощью оператора `instanceof` (который возвращает имя функции, использованной для создания объекта) для поддержки работоспособности механизма перегрузки методов. Фактически, единственным местом, в котором мы не можем успешно провести транскомпиляцию перегружаемых функций, являются участки кода, на которых в функциях используется одинаковое количество аргументов разных числовых типов. Так как язык JavaScript имеет только один числовой тип, объявления таких функций, как `add(int x, int y)`, `add(float x, float y)` и `add(double x, double y)`, будут конфликтовать друг с другом. Во всех других случаях, однако, механизм перегрузки методов работает отлично.

Язык Java позволяет осуществлять импорт скомпилированного кода.

Иногда функций языка программирования Processing становится недостаточно и дополнительные функции могут быть получены из библиотеки функций Processing. Она реализована в форме архива с расширением `.jarchive` и скомпилированным Java-кодом внутри и предоставляет в распоряжение разработчика такие дополнения, как функции для работы с сетью, обработки аудио- и видеоданных, взаимодействия с аппаратным обеспечением, а также другие экзотические функции, не реализуемые самим языком программирования Processing.

Это является проблемой, так как скомпилированный Java-код является байткодом, используемым виртуальной машиной Java. Данное обстоятельство доставило нам много головной боли: как реализовать импорт функций из библиотек без разработки декомпилятора байткода Java? После длившихся практически в течение года дискуссий мы пришли к по-видимому наиболее простому решению. Вместо того, чтобы пытаться также добавить поддержку библиотек языка программирования Processing, мы решили добавить поддержку ключевого слова `import` в скетчи и создать API библиотеки дополнительных функций `Processing.js`, предоставив таким образом разработчи-

кам возможность создания версий их библиотек на языке JavaScript (в том случае, когда эта задача выполнима в условиях работы веб-приложений), таким образом, в том случае, если они разработают библиотеку дополнительных функций, используемую с помощью директивы `import prosessing.video`, язык программирования Processing будет использовать архив с расширением `.jarchive`, а библиотека `Processing.js` вместо этого будет использовать код из файла `processing.video.js`, поэтому в обоих случаях приложение будет "просто работать". Эти функции предназначены для включения в релиз `Processing.js` 1.4, а возможность импорта кода библиотек является последней важной функцией, которой не хватало в `Processing.js` (на данный момент мы поддерживаем ключевое слово `import`, но только таким образом, что оно удаляется из исходного кода перед его преобразованием) и будет последним важным шагом для достижения совместимости.

Для чего использовать язык JavaScript, если он не совместим с Java?

Это не бессмысленный и имеющий множество вариантов ответа вопрос. Наиболее очевидный ответ заключается в том, что интерпретатор языка JavaScript поставляется в составе браузера. Вам не придется "устанавливать" программные компоненты для поддержки языка JavaScript самостоятельно, не требуется плагина для загрузки перед использованием приложений; все необходимое уже здесь. Если вы хотите портировать какое-либо приложение для работы в веб-окружении, вам придется столкнуться с языком JavaScript. При этом, учитывая гибкость языка JavaScript, выражение "столкнуться" на самом деле никак не говорит о том, насколько мощным является язык. Таким образом, одной из причин для выбора языка JavaScript является то, что "этот язык уже здесь". Практически любое интересующее нас устройство на сегодняшний день поставляется с поддерживающим языком JavaScript браузером. Это утверждение не справедливо в случае языка Java, который все реже и реже распространяется в форме предустановленного программного компонента в том случае, если он вообще доступен.

Однако, корректный ответ заключается не в том, что язык JavaScript "не может" выполнять те функции, которые выполняет язык Java; он может выполнять их, но при этом будет работать медленнее. Даже с учетом того, что изначально язык JavaScript не поддерживает некоторые возможности языка Java, следует помнить о том, что он является Тьюринг-полным языком программирования и может эмулировать работу любого другого языка программирования со снижением скорости работы. Технически мы могли бы разработать завершенную реализацию интерпретатора языка Java с наборами объектов `String`, разделенными моделями переменных и методов, ориентацией на объекты, классы и их экземпляры со строгими иерархиями классов и любыми другими функциями, существующими под солнцем и реализованными работниками компании Sun (или, на сегодняшний день, компании Oracle), но это не то, для чего мы хотим использовать его: библиотека `Processing.js` предназначена для преобразования кода Processing в характерный для веб-окружения код с применением такого малого объема вспомогательного кода, как это необходимо. Это значит, что хотя мы и приняли решение о поддержке некоторых специфичных для языка Java возможностей, наша библиотека имеет одно весомое преимущество: она работает со встроенным в страницы кодом на языке JavaScript очень и очень хорошо.

Фактически во время встречи между разработчиками проектов `Processing.js` и `Processing` в помещении компании `Wacom`, расположенной в Бостоне, в 2010 году Ben Fry спросил John Resig о том, почему он использует замену с помощью регулярных выражений и только частичное преобразование кода вместо создания системы разбора кода и компилятора. Ответ от John заключался в том, что для него важно сохранение возможности смешивания пользователями синтаксиса `Processing` (Java) и JavaScript без необходимости выбора одного из них. Этот изначальный выбор был ключевым для формирования философии проекта `Processing.js`. Мы выполнили большой объем работы для предоставления возможности использования этого подхода в рамках нашего кода и четко видим результат этой работы при рассмотрении работ всех создателей "классических веб-приложений", которые используют библиотеку `Processing.js` и никогда не использовали язык про-

граммирования Processing, при этом успешно смешивая синтаксические конструкции языков Processing и JavaScript без лишних проблем.

В следующем примере показано, как может функционировать код со смешанными синтаксическими конструкциями из языков JavaScript и Processing.

```
// Код на языке JavaScript (должна быть сгенерирована ошибка при использовании в
Processing)
var cs = { x: 50,
            y: 0,
            label: "my label",
            rotate: function(theta) {
                var nx = this.x*cos(theta) - this.y*sin(theta);
                var ny = this.x*sin(theta) + this.y*cos(theta);
                this.x = nx; this.y = ny; }};

// Код на языке Processing
float angle = 0;

void setup() {
    size(200,200);
    strokeWeight(15); }

void draw() {
    translate(width/2,height/2);
    angle += PI/frameRate;
    while(angle>2*PI) { angle-=2*PI; }
    jQuery('#log').text(angle); // Код на языке JavaScript (ошибка при использова-
ниии в Processing)
    cs.rotate(angle);           // Корректный код как на языке JavaScript, так и на
языке Processing
    stroke(random(255));
    point(cs.x, cs.y); }
```

Многие вещи в языке Java являются обещаниями: строгая типизация является обещанием для компилятора, относящимся к данным, область видимости является обещанием того, кто будет вызывать методы и ссылаться на переменные, интерфейсы являются обещаниями о том, что экземпляры классов содержат методы, описанные в рамках интерфейса, и т.д. Нарушение этих обещаний приведет к жалобам компилятора. Но в том случае, если вы не нарушаете их, что является наиболее важным аспектом архитектуры библиотеки Processing.js, вам не понадобится дополнительного кода для выполнения этих обещаний, чтобы программа работала. Если вы устанавливаете числовое значение переменной и ваш код рассматривает эту переменную как числовую, то в конце концов объявление `var varname` ничем не хуже объявления `int varname`. Вам необходима типизация? При использовании языка Java это так; в случае использования JavaScript она не требуется, поэтому зачем принуждать разработчиков использовать ее? Аналогичный подход используется в отношении других обещаний. Если компилятор языка Processing не жалуется на ваш код, мы можем убрать все точные синтаксические конструкции, соответствующие описанным обещаниям, и код все так же будет работоспособен.

Этот подход сделал библиотеку Processing.js чрезвычайно часто используемой при создании визуализаций данных, мультимедийных презентаций и даже развлекательных приложений. Скетчи, созданные с использованием классического синтаксиса языка Processing, работают, при этом скетчи, в которых смешивается синтаксис языков Java и JavaScript, также отлично работают, как впрочем и скетчи, при создании которых используется классический синтаксис языка JavaScript и которые рассматривают библиотеку Processing.js в качестве улучшенного фреймворка для рисования на канве. После приложения участниками проекта усилий к созданию альтернативы классическому языку программирования Processing без жесткого требования исключительного использования синтаксиса языка Java, проект стал использоваться широким кругом пользователей, представлен-

ным в масштабе всей сферы веб-технологий. Мы наблюдали примеры использования библиотеки Processing.js в масштабе всей глобальной сети. Все компании, начиная с IBM и заканчивая Google, создавали приложения для визуализации, приложения для создания презентаций и даже простые игры с помощью Processing.js - библиотека Processing.js набирала вес.

Другой замечательной особенностью процесса преобразования кода, использующего синтаксис языка Java в код, использующий синтаксис языка JavaScript, при условии невмешательства в уже существующий код на языке JavaScript является то, что мы реализовали возможность, о которой даже не думали: библиотека Processing.js получила возможность работы с любыми приложениями, которые используют язык JavaScript. Одной из действительно интересных вещей, которую мы наблюдаем на сегодняшний день, например, является то, что люди используют язык CoffeeScript (изящно упрощенный язык программирования похожий на Ruby, в ходе работы которого происходит транскомпиляция кода в код на языке языке JavaScript), комбинируя его с библиотекой Processing.js, и добиваются по истине превосходных результатов. Даже если бы мы намеревались создать "язык программирования Processing для веб-приложений", осуществляющий разбор синтаксических конструкций языка Processing, люди воспользовались бы результатами нашей работы и добавили поддержку совершенно новых синтаксисов. Но они никогда не сделали бы этого в том случае, если бы мы реализовали библиотеку Processing.js в форме простого интерпретатора Java-кода. Используя преобразование кода вместо реализации его интерпретатора, библиотека Processing.js поспособствовала широкому распространению языка Processing в сфере веб-приложений в более значительной степени, чем это могло бы случиться при исключительном использовании синтаксических конструкций языка Java, или даже в том случае, если бы использовался исключительно синтаксис языка Java, но выполнение приложений осуществлялось с привлечением средств языка JavaScript. Использование нашего кода не только конечными пользователями, но и разработчиками, пытающимися интегрировать его со своими технологиями, выглядело удивительно и вдохновляюще. Было ясно то, что мы делаем что-то правильно и сообщество веб-разработчиков довольно результатами нашей работы.

Результат

В преддверии релиза библиотеки Processing.js версии 1.4.0 следует отметить то, что результатом нашей работы уже является библиотека, с помощью которой можно выполнить любой переданный скетч при том условии, что он не импортирует функций из скомпилированных Java-библиотек. Если вы можете разработать приложение на языке Processing, и оно будет корректно функционировать, вы сможете также разместить его на веб-странице и просто запустить его. Из-за различий в методах доступа к аппаратному обеспечению и в низкоуровневых реализациях различных частей конвейера обработки изображений, появятся различия во временных интервалах, но в общем случае скетч, который выводит изображение с частотой 60 кадров в секунду при работе в Processing IDE будет выводить изображение с такой же частотой 60 кадров в секунду при работе на современном компьютере с современным браузером. Мы достигли точки развития проекта, в которой число сообщений об ошибках начало сокращаться и большая часть работы заключается не в добавлении новых возможностей, а в исправлении и оптимизации кода.

Благодаря усилиям множества разработчиков, работающих над исправлением более 1800 ошибок, описанных в сообщениях пользователей, скетчи на языке Processing "просто работают" при использовании библиотеки Processing.js. Даже те скетчи, которые импортируют код библиотек могут работать в том случае, если в распоряжении имеется исходный код используемой библиотеки. При благоприятных обстоятельствах библиотека может быть разработана таким образом, что у вас будет возможность преобразовать ее к виду, использующему классический синтаксис языка Processing, путем выполнения нескольких операций поиска и замены строк. В этом случае код может начать работу в виртуальном пространстве немедленно. В том же случае, когда библиотека предоставляет такие функции, которые не могут быть реализованы с использованием классического синтаксиса языка Processing, но могут быть реализованы с использованием классического синтаксиса языка JavaScript, потребуется дополнительная работа для эффективной эмуляции би-

лиотеки с помощью кода на языке JavaScript, но портирование все так же возможно. Единственными функциями, при использовании которых код на языке Processing не может быть портирован, являются функции, по своей сути не доступные для браузеров, такие, как функции для непосредственного взаимодействия с аппаратным обеспечением (таким, как веб-камеры или платы Arduino) или функции, выполняющие необрабатываемые операции записи данных на диск, хотя даже эта ситуация меняется. Браузеры постоянно расширяют свои функции для возможности выполнения все более сложных приложений и актуальные на сегодняшний день сдерживающие факторы через год могут исчезнуть, таким образом, можно надеяться на то, что в не очень далеком будущем даже скетчи, которые на данный момент невозможны запустить в браузере, смогут быть портированы.

17.3. Компоненты кода

Библиотека Processing.js распространяется и разрабатывается в виде одного большого файла, но в плане архитектуры она может быть разделена на три отдельных компонента: 1) загрузчик, ответственный за преобразования исходного кода на языке Processing в исходный код на языке JavaScript с синтаксическими конструкциями из Processing.js и его исполнение, 2) статические функции, которые могут быть использованы всеми скетчами и 3) функции скетчей, которые должны ставиться в соответствие их отдельным экземплярам.

Загрузчик

Загрузчик является программным компонентом, управляющим тремя процессами: процессом предварительной обработки кода, процессом преобразования кода и процессом исполнения скетча.

Предварительная обработка кода

На этапе предварительной обработки кода директивы библиотеки Processing.js выделяются из кода и обрабатываются. Эти директивы могут быть разделены на два типа: установки и инструкции загрузки. Существует небольшое количество директив, оформленных в соответствии с философией "простой работы" и единственный тип установок, которые могут быть изменены авторами скетчей, относятся к взаимодействию со страницей. По умолчанию скетч будет выполняться в том случае, когда страница не находится в фокусе, но директива `pauseOnBlur = true` устанавливает параметры скетча таким образом, что он будет прерывать исполнение в моменты, когда страница, на которой располагается скетч, уходит из фокуса и восстанавливать работу в момент, когда страница вновь появляется в фокусе. Также по умолчанию клавиатурный ввод перенаправляется скетчу только в том случае, если он находится в фокусе. Это особенно важно в том случае, когда люди выполняют множество скетчей на одной и той же странице, так как клавиатурный ввод, предназначенный для одного из скетчей, не должен обрабатываться другим. Однако, эта функция может быть отключена с помощью директивы `globalKeyEvents = true`, в результате чего все события от клавиатуры будут передаваться каждому из скетчей, выполняемых на странице.

Инструкции загрузки выполняются в форме вышеописанных предварительных загрузок изображений и шрифтов. Так как изображения и шрифты могут использоваться множеством скетчей, они загружаются и отслеживаются глобально, поэтому различные скетчи не будут пытаться загрузить несколько раз один и тот же ресурс.

Преобразование кода

Компонент преобразования кода формирует ветви дерева абстрактного синтаксического анализа на основе таких элементов исходного кода, как выражения, методы, переменные, классы, и.т.д. После этого данное дерево абстрактного синтаксического анализа преобразуется в исходный код на языке JavaScript, из которого в процессе исполнения формируется эквивалентная скетчу про-

грамма. Этот преобразованный исходный код максимально использует экземпляр фреймворка Processing.js для установления отношений классов, причем классы из исходного кода на языке Processing преобразуются в прототипы языка JavaScript со специальными функциями для установления родительских классов и объединениями с функциями и переменными родительских классов.

Исполнение скетча

Финальным этапом процесса загрузки является исполнение скетча, которое начинается с установления того, завершена ли его предварительная загрузка и, в случае ее завершения, продолжается путем добавления скетча в список исполняемых приложений и генерации в рамках скетча характерного для языка JavaScript события с именем `onLoad`, таким образом, все обработчики событий скетча смогут выполнить необходимые действия. После этого начинается исполнение цепочки функций приложения Processing: сначала вызывается функция `setup`, а затем - `draw` и в том случае, если скетч исполняется в цикле, устанавливается интервал времени для вызовов функции `draw`, причем длительность этого интервала выбирается с учетом максимального приближения частоты вывода изображения к желаемой частоте кадров для скетча.

Статическая библиотека

Большая часть работы библиотеки Processing.js выполняется в рамках "статической библиотеки", которая предоставляет описания констант, универсальных функций и универсальных типов данных. Большая часть этих элементов выполняет двойную работу, так как они описаны в виде глобальных свойств, но при этом также на них приведены ссылки из экземпляров классов для ускорения исполнения кода. Такие глобальные константы, как коды ключей и обозначения цветов размещены в самом объекте языка Processing, их значения устанавливаются однократно, после чего используются ссылки из объектов, созданных с использованием конструктора языка Processing. Такой же подход, позволяющий разрабатывать код в максимальном соответствии с принципом "однократной разработки и повсеместного запуска" без снижения производительности, используется для работы с самостоятельными вспомогательными функциями.

Библиотека Processing.js должна поддерживать большое количество сложных типов данных не только для того, чтобы реализовывать поддержку типов из языка программирования Processing, но также и для осуществления внутренних операций. Эти типы данных также описаны в рамках конструктора Processing:

- `Char`, внутренний объект, используемый для совместимости с некоторыми особенностями типа данных `char` языка программирования Java.
- `PShape`, представляющий объекты контуров.
- `PShapeSVG`, расширение для объектов `PShape`, созданное для представления контуров в формате SVG XML. Для функционирования объекта `PShapeSVG` мы реализовали код, осуществляющий формирование инструкций преобразования из формата SVG в формат данных элемента `<canvas>`. Так как язык программирования Processing не реализует полной поддержки формата SVG, мы создали код, не опираясь на функции каких-либо сторонних библиотек для работы с файлами SVG, что подразумевает возможность учета каждой строки кода, относящейся к функции импорта данных в формат SVG. Этот код всего лишь производит анализ необходимых данных и не занимает лишнее место из-за того, что не реализует неподдерживаемые в рамках классического языка Processing функции в четком соответствии со спецификацией.
- `XMLElement`, объект документа XML. Для функционирования объекта `XMLElement` мы также реализовали наш собственный код на основе функций браузера, предназначенный для первоначальной загрузки элемента XML в структуру представления ветвей (Node) и последующего обхода этой структуры для формирования упрощенного объекта представления документа. Это также значит, что в библиотеке Processing.js нет неиспользуемого кода, занимающего место и потенциально приводящего к ошибкам после применения исправления, использующего некорректную функцию.
- `PMatrix2D` и `PMatrix3D`, которые выполняют матричные операции в двумерных и трехмерных режимах.
- `PImage`, который представляет ресурс изображения. На самом деле это просто обертка над объектом `Image` с некоторыми дополнительными функциями и свойствами для соответствия API объекта API, используемому языком программирования Processing.
- `PFont`, представляющий ресурс шрифта. Объекта для работы со шрифтом `Font`, объявленного в рамках языка JavaScript, не существует (во всяком случае, на дан-

ный момент), поэтому вместо хранения шрифта в виде объекта, наша реализация `PFont` загружает шрифт средствами браузера, вычисляет его метрики на основе текста, выводимого браузером при использовании данного шрифта, после чего кэширует результирующий объект `PFont`. Для повышения скорости работы объект `PFont` содержит ссылку на канву, которая использовалась для установления свойств шрифта с целью использования в случае, когда должно быть вычислено значение свойства `textWidth`, но из-за того, что регистрация объектов `PFont` реализуется на основе пары имя/размер, в случае использования скетчом множества шрифтов разных размеров или вообще множества шрифтов, значительно увеличивается потребление памяти. По этой причине каждый из объектов `PFont` будет очищать свою кэшированную канву и вместо ее использования вызывать стандартную функцию вычисления значения `textWidth` в том случае, если кэш значительно увеличивается в размере. Вторая стратегия сохранения памяти заключается в том, что в случае продолжения увеличения размера кэша после очистки каждой кэшированной канвы объекта `PFont`, кэширование шрифтов полностью отключается и изменения шрифтов скетча приводят к созданию новых впоследствии отбрасываемых объектов `PFont` в случае любого изменения имени шрифта, размера текста или самого текста.

- `DrawingShared`, `Drawing2D` и `Drawing3D`, которые реализуют все функции для работы с графикой. Объект `DrawingShared` в наибольшей степени оказывает воздействие на скорость работы библиотеки `Processing.js`. Он устанавливает, работает ли скетч в двумерном или в трехмерном режиме вывода графики, после чего связывает все функции для работы с графикой либо с функциями объекта `Drawing2D`, либо с функциями объекта `Drawing3D`. После этого можно быть уверенным в том, что будет использован кратчайший путь кода для выполнения инструкций вывода графики, а также скетчи, работающие в двумерном режиме, не будут использовать функции для работы с трехмерной графикой и наоборот. Связывая функции обработки графики только с одним из двух наборов функций, мы повышаем скорость работы приложения, так как в этом случае не приходится в рамках каждой функции устанавливать режим вывода графики и выбирать путь исполнения кода, что в свою очередь позволяет нам сократить объем кода, не связывая функции, которые гарантированно не будут использоваться.
- `ArrayList`, контейнер, эмулирующий поведение объекта `ArrayList` из Java.
- `HashMap`, контейнер, эмулирующий поведение объекта `HashMap` из Java.

`ArrayList` и `HashMap` в частности, являются специальными структурами данных при рассмотрении их реализации в рамках языка Java. Эти контейнеры функционируют с использованием концепций языка Java о эквивалентности и хэшировании, при этом все объекты языка Java содержат методы `equals` и `hashCode`, позволяющие хранить их в списках и таблицах.

В случае использования контейнеров без хэширования, поиск объектов осуществляется на основе эквивалентности, а не идентичности. Следовательно, вызов `list.remove(myobject)` приводит к итерационному обходу списка в поисках элемента, для которого вызов метода `element.equals(myobject)` вместо равенства `element == myobject` возвращает истинное логическое значение. Из-за того, что все объекты должны реализовывать метод `equals`, мы реализовали "виртуальную функцию `equals`" на уровне языка JavaScript. Эта функция принимает два объекта в качестве аргументов, проверяет, реализует ли каждый из них свою функцию `equals` и в том случае, если это так, использует эти функции. Если же функции не реализованы, а переданные объекты являются примитивами, выполняется проверка эквивалентности примитивов. В противном случае эти объекты считаются неэквивалентными.

В случае использования хэширующих контейнеров ситуация становится еще интереснее, так как хэширующие контейнеры работают по принципу деревьев, сформированных из ссылок. На самом деле контейнер состоит из динамически меняющегося количества списков, каждый из которых поставлен в соответствие определенному хеш-коду. Поиск объектов начинается с поиска контейнера, хеш-код которого совпадает с искомым кодом, после чего поиск объекта осуществляется в рамках найденного контейнера путем установления эквивалентности объектов. Так как все объекты языка Java реализуют метод `hashCode`, мы также разработали "виртуальную функцию вычисления хеш-кода", которая принимает единственный объект в качестве аргумента. Функция проверяет, реализует ли объект свою функцию `hashCode` и в том случае, если это так, использует эту функцию. В противном случае хеш-код вычисляется с помощью того же алгоритма хэширования, который используется языком Java.

Администрирование

Последней функцией библиотеки статического кода является поддержание в актуальном состоянии списка выполняющихся на данный момент на странице экземпляров скетчей. Список экземпляров скетчей формируется на основе идентификаторов канвы, используемой при загрузке каждого из скетчей, поэтому пользователи могут осуществить вызов

`Processing.getInstanceById('идентификатор_канвы')` и получить ссылку на свой скетч для взаимодействия с ним.

Код экземпляров классов

Код экземпляров классов разрабатывается в форме объявлений `p.functor = function(arg, _)` для API языка `Processing`, и в форме `p.constant = _` для переменных состояния скетча (где `p` является установленной ссылкой на скетч). Ни одно из этих объявлений не находится в отдельных блоках кода. Наоборот, код организован на основе функций, поэтому код, реализующий операции с экземплярами классов `PShape` размещен в непосредственной близости к объявлению объекта

PShape и код для осуществления графических операций с участием экземпляров классов размещается недалеко или в самих объявлениях объектов Drawing2D и Drawing3D.

Для повышения быстродействия большая часть кода, который мог бы быть разработан в форме статического кода с оберткой в рамках экземпляра класса, реализуется исключительно в форме кода экземпляра класса. Например, функция `lerpColor(c1, c2, ratio)`, которая устанавливает цвет, относящийся к процессу работы алгоритма линейной интерполяции двух цветов, объявляется в форме функции экземпляра класса. Вместо того, чтобы использовать вызов `p.lerpColor(c1, c2, ratio)`, являющийся оберткой над какой-либо статической функцией `Processing.lerpColor(c1, c2, ratio)`, в том случае, когда фактически никакая из других частей библиотеки Processing.js не использует функцию `lerpColor`, код будет выполняться быстрее в том случае, если он будет представлен в форме функции экземпляра класса. Хотя такой подход и увеличивает размер описания класса, большая часть функций, в отношении которых у нас могло бы возникнуть желание о преобразовании их в функции экземпляров классов вместо использования обертки для функции статической библиотеки, представлена функциями малого размера. Следовательно, увеличивая потребление памяти, мы создаем действительно быстрые пути исполнения кода. В то время, как для всего кода объекта Processing при запуске одномоментно резервируется участок памяти размером в 5 МВ, необходимый для работы отдельных скетчей код занимает около 500 КВ памяти.

17.4. Разработка библиотеки Processing.js

Разработка библиотеки Processing.js ведется в интенсивном темпе в первую очередь из-за того, что процесс разработки подчиняется нескольким основным правилам. Так как эти правила влияют на архитектуру библиотеки Processing.js, стоит кратко рассмотреть их перед завершением этой главы.

Создавать работающий код

Выражение "создание работающего кода" может показаться тавтологическим; вы разрабатываете код, и в результате ваш код либо работает, так как вы именно для этого его и разрабатывали, либо не работает, что подразумевает то, что вы пока не справились с поставленной задачей. Однако, выражение "создание работающего кода" может быть записано в развернутой форме следующим образом: "создание работающего кода и предоставление доказательств его работоспособности после завершения процесса разработки".

Наряду с многими особенностями процесса разработки, существует одна особенность, позволившая библиотеке Processing.js поддерживать такие высокие темпы развития и заключающаяся в наличии тестов. Любое сообщение об ошибке, требующее вмешательства в код либо путем разработки нового кода, либо путем исправления старого кода, не может быть помечено как исправленное до того момента, пока не будет предоставлен модульный или сравнительный тест, позволяющий сторонним людям проверить не только то, что код начал работать так, как должен, но и то, что код может работать некорректно при определенных изменениях. Для большей части кода в этом случае обычно разрабатывается модульный тест - небольшой фрагмент кода, который вызывает функцию и просто проверяет, возвращает ли функция корректные значения в корректных и некорректных случаях вызова функции. Этот подход позволяет нам не только тестировать присылаемый код, но и производить тестирование с целью поиска регрессий.

Перед тем, как какой-либо код принимается и переносится в стабильную разрабатываемую ветку исходного кода, модифицированная библиотека Processing.js подвергается тестированию с использованием постоянно расширяющегося набора модульных тестов. Значительные исправления и тесты производительности, в частности, обуславливают использование отдельного набора модульных тестов, в отличие от случая с поиском элементов библиотеки, которые работали корректно до внесения изменений. Наличие теста для каждой функции в рамках API библиотеки наряду с тестами для внутренних функций подразумевает то, что по мере развития библиотеки Processing.js

не произойдет внезапного нарушения совместимости с предыдущими версиями. Запрет деструктивных изменений API библиотеки заключается в том, что если ни один из тестов не закончился неудачей перед внесением нового или модификаций старого кода, ни один из тестов также не должен закончиться неудачей в случае использования измененного кода.

Ниже приведен пример модульного теста для проверки внутреннего процесса создания объекта.

```
interface I {
    int getX();
    void test(); }

I i = new I() {
    int x = 5;
    public int getX() {
        return x; }
    public void test() {
        x++; }};

i.test();

_checkEqual(i.getX(), 6);
_checkEqual(i instanceof I, true);
_checkEqual(i instanceof Object, true);
```

В дополнение к регулярному использованию модульных тестов, мы также проводим сравнительное визуальное тестирование (или "ref"-тесты). Так как библиотека Processing.js представляет собой порт языка программирования, предназначенного для визуального представления данных, некоторые типы тестирования не могут осуществляться исключительно с помощью модульных тестов. Тестирование с целью установления того, использованы ли корректные пиксели для изображения эллипса, или изображена ли вертикальная линия толщиной в один пиксель четко или с использованием сглаживания, не может быть проведено без визуального сравнения. Так как все распространенные браузеры реализуют элемент <canvas> и API Canvas2D с небольшими различиями, эти вещи могут быть протестированы только путем запуска кода в браузере и проверки того, что результирующее изображение, сформированное скетчем, выглядит аналогично изображению, сгенерированному при использовании языка программирования Processing. Для упрощения жизни разработчиков мы используем для этого набор автоматических тестов, в котором новые варианты тестов запускаются с использованием языка программирования Processing для генерации "эталонных" данных, которые впоследствии будут использоваться для попиксельного сравнения. После этого описанные данные сохраняются в форме комментария в скетче, с помощью которого они генерировались, формируя тест, а сами эти тесты впоследствии выполняются с использованием библиотеки Processing.js на странице для визуального сравнительного тестирования, причем при выполнении каждого теста производится попиксельное сравнение между тем "как должен выглядеть результат теста" и тем, "как он выглядит в реальности". Если значения пикселей отличаются, тест завершается неудачей и разработчику предоставляются три изображения: как должен выглядеть результат теста, как результат был сформирован средствами библиотеки Processing.js и изображение с указанием различий между двумя изображениями путем обозначения проблемных областей с помощью пикселей красного цвета, а областей корректных данных - белого. Как и в случае модульных тестов, эти тесты должны завершиться успешно перед приемом любого стороннего кода.

Создавать быстрый код

В ходе работы над проектом с открытым исходным кодом, создание работоспособной функции является всего лишь первым шагом в рамках ее жизненного цикла. Как только вы добиваетесь работоспособности функции, появляется желание проверки того, что она работает быстро. Основываясь на принципе, формулируемом следующим образом: "если вы не сможете проверить быстро-

действие функции, вы не сможете усовершенствовать ее", большая часть функций из состава библиотеки Processing.js снабжается не только ref-тестами, но и тестами производительности (или "perf"-тестами). Небольшие фрагменты кода, которые просто вызывают функцию без проверки корректности ее работы, выполняются по несколько сотен раз подряд, причем время их выполнения сохраняется на специальной странице тестирования производительности. Это позволяет оценить, насколько хороша (или плоха!) производительность библиотеки Processing.js в браузерах, поддерживающих элемент <canvas> из состава HTML5. Всегда после прохождения патчем для оптимизации модульного и сравнительного тестирования, он также проходит тестирование с помощью нашей страницы тестирования производительности. JavaScript является любопытным языком и замечательный код фактически может работать в несколько раз медленнее, чем код, содержащий большее количество тех же строк, оформленных в виде внутреннего кода вместо отдельной вызываемой функции. Это делает процесс тестирования производительности чрезвычайно важным. Нам удалось повысить производительность определенных частей библиотеки в три раза, просто обнаружив часто используемые циклы в ходе тестирования производительности и сократив количество вызовов функций путем формирования внутреннего кода, а также путем преобразования функций для возврата значений в тот момент, когда они становятся известны вместо возврата значения в самом конце функции.

Другим способом, которым мы пытаемся увеличить производительность библиотеки Processing.js, является исследование возможностей ее рабочего окружения. Так как библиотека Processing.js жестко зависит от производительности интерпретаторов JavaScript, имеет смысл также рассмотреть то, какие возможности различные интерпретаторы предоставляют для ускорения работы сценариев. Это актуально особенно сегодня, когда браузеры начинают поддерживать функции аппаратного ускорения графических операций, ведь в том случае, если интерпретаторы предоставляют новые и более эффективные типы данных и функции для выполнения необходимых для работы Processing.js низкоуровневых операций, становятся возможными мгновенные увеличения производительности. Например, язык JavaScript по техническим причинам не использует статическую типизацию, но окружения для разработки приложений, взаимодействующих с аппаратным обеспечением для работы с графикой, используют ее. Раскрывая используемые для непосредственного взаимодействия с аппаратным обеспечением структуры данных на уровне языка JavaScript, становится возможным значительное повышение производительности фрагментов кода в том случае, если нам известно то, при их работе будут использоваться определенные значения.

Создавать краткий код

Существует два способа сокращения объема кода. Во-первых, это разработка компактного кода. Если вы осуществляете многократные манипуляции с переменной, постарайтесь сократить их в одну операцию (если это возможно). В том случае, если вы осуществляете многократный доступ к переменной объекта, осуществляйте кэширование. Если вы вызываете функцию несколько раз, кэшируйте результат. Осуществляйте выход из функции как только вы получаете всю необходимую информацию и в общем случае применяйте все приемы, которые применил бы оптимизатор кода, самостоятельно. Определенно, язык программирования JavaScript замечательно подходит для этого, так как он очень гибок. Например, вместо использования конструкции:

```
if ((result = functionresult) !== null) {
    var = result;
} else {
    var = default;
}
```

В JavaScript можно использовать:

```
var = functionresult || default
```

Также существует другая форма увеличения компактности кода, которая связана с процессом его исполнения. Так как язык JavaScript позволяет вам изменять связи функций в процессе работы приложения, выполняемый код значительно уменьшается в размере в том случае, если вы скажете: "привязать функцию рисования двумерных линий к вызову функции line" сразу после того, как узнаете, что программа работает в двумерном, а не в трехмерном режиме, следовательно, вам не придется использовать условный переход

```
if(mode==2D) { line2D() } else { line3D() }
```

в каждой функции, которая может работать как в двумерном, так и в трехмерном режиме.

Наконец, существует процесс минимизации объема кода. Существует множество качественных систем, которые позволяют сжать ваш код на языке JavaScript путем переименования переменных, удаления пробелов и применения определенных оптимизаций кода, которые сложно произвести вручную при условии его последующей читаемости. Примерами таких систем являются YUI minifier и Google closure compiler. Мы используем эти технологии в рамках библиотеки Processing.js для увеличения пропускной способности каналов пользователей - минимизация кода, достигаемая путем удаления комментариев, соответствует уменьшению размера библиотеки на целых 50%, при этом в случае использования преимущества процесса взаимодействия современных серверов и браузеров путем передачи сжатых с помощью gzip данных, мы можем получить полнофункциональную сжатую библиотеку Processing.js в размером в 65 KB.

Если ничего не получается, обратиться к людям

Не все функции, реализованные на данный момент в языке Processing могут работать в браузере. Модели систем безопасности предотвращают выполнение определенных действий, таких, как сохранение файлов на жесткий диск и осуществление операций ввода/вывода с использованием последовательных портов и портов USB, также отсутствие типизации в языке JavaScript может привести к непредсказуемым последствиям (таким, как выполнение всех математических действий с использованием чисел с плавающей точкой). Иногда мы сталкивались с выбором между вариантом добавления огромного объема кода для работы в частном случае и вариантом добавления отметки "невозможно исправить" к сообщению об ошибке. В таких случаях создавалось новое сообщение об ошибке, обычно с названием "Добавить документацию, в которой приводится объяснение причин...".

Для того, чтобы не забывать об этих случаях, у нас есть документация, предназначенная для людей, начавших использовать библиотеку Processing.js и обладающих опытом работы с языком программирования Processing, а также для людей, которые начали использовать библиотеку Processing.js, обладая опытом работы с языком программирования JavaScript, охватывающая различия между тем, что ожидается и тем, что на самом деле происходит. Некоторые особенности просто заслуживают отдельного упоминания, так как вне зависимости от того, сколько усилий мы приложили к разработке библиотеки Processing.js, остаются функции, которые мы не можем добавить в библиотеку без ущерба пользовательским качествам. Хорошая архитектура затрагивает не только вопросы о том, как реализуются функции, но и вопросы о том, почему они так реализуются; без ответов на эти вопросы у вас будут возникать одни и те же дискуссии на тему того, почему код выглядит определенным образом и должен ли он выглядеть по-другому каждый раз после смены команды разработчиков.

17.5. Выученные уроки

Наиболее важный выученный нами в ходе разработки библиотеки Processing.js урок заключается в том, что при портировании языка программирования важным является то, что в результате приложения работают корректно, а не то, идентичен ли используемый вашим портом код оригиналу.

Несмотря на то, что синтаксис языков Java и JavaScript чрезвычайно похож и преобразование кода на языке Java в код на языке JavaScript достаточно просто, обычно следует обращать внимание на то, что можно сделать средствами самого языка JavaScript для получения того же функционального результата. Использование преимущества отсутствия типизации данных путем повторного использования переменных, использование определенных встроенных функций, которые быстры в JavaScript, но медленны в Java или пренебрежение шаблонами программирования, которые быстры в Java, но медленны в JavaScript, может привести к тому, что ваш код будет радикально отличаться, но работать абсолютно так же. Вам часто приходилось слышать от людей предостережения об изобретении колеса, но эти слова относятся только к тем случаям, когда используется один и тот же язык программирования. При портировании же следует изобретать столько колес, сколько понадобится для достижения требуемой производительности.

Другой важный урок заключается в том, что следует всегда завершать выполнение функций так рано, как это возможно и максимально сократить количество ветвлений. Блок if/then с последующим оператором возврата значения функции return может работать (иногда значительно) быстрее в случае использования вместо него конструкции if-return/return, причем в ней оператор возврата значения будет использоваться как условная ссылка. Хотя концептуально верным решением и является полное установление состояния функции перед возвратом результата выполнения этой функции, это также означает, что есть вероятность выполнения кода, который никак не относится к возвращаемому значению. Не теряйте циклы ЦП; возвращайте результат выполнения функции сразу после того, как у вас в распоряжении находится вся необходимая информация.

Третий урок касается тестирования вашего кода. В начале разработки библиотеки Processing.js мы воспользовались преимуществом наличия очень качественной документации, описывающей то, как язык программирования Processing "должен" работать, а также большого набора тестов, большая часть из которых в то время "неудачно выполнялась по известным причинам". Это обстоятельство позволило нам сделать две вещи: 1) разрабатывать код для прохождения тестов и 2) создавать тесты перед разработкой кода. В ходе обычного процесса разработки, когда разрабатывается код вместе с соответствующими тестами, на самом деле создаются предвзятые тесты. Вместо проверки того, выполняет ли ваш код необходимые действия в соответствии со спецификацией, вы проверяете только то, не содержит ли ваш код ошибок. При разработке библиотеки Processing.js мы начали создавать тесты, основываясь не на том, какие требования предъявляются к определенной функции или набору функций, а на документации для них. Обладая этими непредвзятыми тестами, мы можем разрабатывать функционально завершенный код вместо просто корректного, но, возможно, не функционирующего кода.

Последний урок является также наиболее общим: применяйте правила гибкой методологии разработки также и к отдельным исправлениям. Ни для кого не будет полезно, если вы начнете разработку и в течении трех дней не будете взаимодействовать с окружающими людьми, прорабатывая идеальное решение. Вместо этого приведите ваше решение в форме кода в работоспособное состояние, причем не обязательно для всех тестовых случаев, и попросите составить отчеты об использовании. Работая в одиночку и применяя набор тестов для устранения ошибок, вы не получаете гарантии того, что будет создан качественный и завершенный код. Никакое количество тестов не поможет вам узнать о том, что вы забыли разработать тесты для определенных частных случаев или о том, что существует лучший алгоритм по сравнению с тем, что вы выбрали, или даже о том, что вам необходимо поменять местами объявления для улучшения подготовки кода к JIT-компиляции. Рассматривайте исправления как релизы: представляйте общественности исправления как можно раньше, обновляйте их как можно чаще и превращайте отчеты об использовании программного компонента в усовершенствования.

18. Puppet

18.1. Введение

Puppet является инструментом для управления ИТ-инфраструктурой, разработанным с использованием языка программирования Ruby и используемым для автоматизации обслуживания данных центров и управления серверами компаний Google, Twitter, Нью-Йоркской фондовой биржи и многих других организаций. Главным образом развитие проекта поддерживается организацией Puppet Labs, которая положила начало его развитию. Puppet может управлять серверами в количестве от 2 до 50,000 и обслуживаться командой, состоящей из одного или сотен системных администраторов.

Puppet является инструментом, предназначенным для настройки и поддержания режима работы ваших компьютеров; используя его простой язык описания конфигурации, вы можете описать для Puppet то, как вы хотите сконфигурировать свои машины, после чего он изменит их конфигурацию в случае необходимости для достижения соответствия вашей спецификации. По мере того, как вы будете изменять эту спецификацию с течением времени под воздействием таких обстоятельств, как обновления пакетов, добавление новых пользователей или обновления конфигурации, Puppet автоматически обновит конфигурацию ваших машин для соответствия спецификации. В случае, если они уже сконфигурированы должным образом, Puppet не будет выполнять никакой работы.

В общем случае Puppet выполняет все возможные действия направленные на то, чтобы использовать функции существующей системы для выполнения своей работы; т.е., в дистрибутивах, основанных на технологиях компании RedHat, он будет использовать утилиту yum для управления пакетами и init.d для управления службами, при этом в операционной системе OS X он будет использовать утилиту dmig для управления пакетами и launchd для управления службами. Одной из основополагающих целей проекта Puppet является выполнение полезной работы вне зависимости от того, используется ли для этого код проекта Puppet или сама система, поэтому следующие системные стандарты являются критичными.

Проект Puppet создан с учетом опыта использования множества других инструментов. В мире приложений с открытым исходным кодом наибольшее влияние на его развитие оказал проект CFEngine, который являлся первым инструментом конфигурации общего назначения с открытым исходным кодом, а также проект ISconf, который использовал утилиту make для выполнения всей работы, что в свою очередь обусловило особое внимание к явно описанным зависимостям в системе. В мире коммерческого программного обеспечения Puppet может рассматриваться как конкурент проектов BladeLogic и Opsware (которые впоследствии были приобретены более крупными компаниями), каждый из которых успешно продавался в момент появления Puppet, но при этом каждый из этих инструментов продавался руководителям больших компаний, вместо развития в соответствии с непосредственными требованиями к качественным инструментам системных администраторов. Предназначением проекта Puppet было решение аналогичных решаемых этими инструментами проблем, при этом он был предназначен для совершенно других пользователей.

В качестве простого примера метода использования Puppet, ниже приведен фрагмент кода, который позволяет быть уверенным в правильной установке и конфигурации службы безопасной оболочки (SSH):

```
class ssh {
    package { ssh: ensure => installed }
    file { "/etc/ssh/sshd_config":
        source => 'puppet:///modules/ssh/sshd_config',
        ensure => present,
        require => Package[ssh]
    }
    service { sshd:
        ensure => running,
        require => [File["/etc/ssh/sshd_config"], Package[ssh]]
    }
}
```

Этот код позволяет быть уверенным в том, что пакет будет установлен, файл будет размещен в необходимом месте и служба будет запущена. Следует отметить, что мы описали зависимости между ресурсами, поэтому всегда будем выполнять любую работу в корректной последовательности. Этот класс может быть ассоциирован с любым узлом для применения заданной конфигурации в рамках этого узла. Обратите внимание на то, что строительными блоками конфигурации Puppet являются структурированные объекты, в данном случае это объекты `package`, `file` и `service`. В терминологии Puppet мы называем эти объекты *ресурсами* (*resources*) и любые спецификации конфигурации Puppet состоят из этих ресурсов и зависимостей между ними.

Нормальная установка Puppet будет содержать десятки или даже сотни этих фрагментов кода, называемых нами классами (*classes*); мы храним эти классы на диске в файлах, называемых манифестами (*manifests*), а также объединяя логически связанные классы в рамках групп, называемых модулями (*modules*). Например, вы можете иметь в распоряжении модуль `ssh` с этим классом `ssh` и любыми другими логически связанными классами наряду с модулями `mysql`, `apache` и `sudo`.

Большая часть операций взаимодействия с Puppet осуществляется с использованием командной оболочки или постоянно работающих служб HTTP, но существуют и графические интерфейсы для выполнения таких задач, как обработка отчетов. Компания Puppet Labs также предоставляет коммерческие программные продукты для работы с Puppet, которые используют графические веб-интерфейсы.

Первый прототип Puppet был разработан летом 2004 года, а полноценная разработка проекта началась в феврале 2005 года. Изначально он был спроектирован и разработан Luke Kanies, системным администратором, имеющим большой опыт разработки небольших инструментов, но не имеющим опыта разработки проектов, содержащих более 10,000 строк кода. По существу Luke Kanies получил навыки программирования в ходе разработки проекта Puppet, что отразилось на архитектуре проекта как положительно, так и отрицательно.

Puppet разрабатывался изначально и в первую очередь как инструмент для системных администраторов, облегчающий их жизнь, позволяющий выполнять работу быстрее, более эффективно и с меньшим количеством ошибок. Первой ключевой инновацией для реализации этого принципа были описанные выше ресурсы, являющиеся примитивами Puppet; они могут переноситься между операционными системами, при этом абстрактно представляя детали реализации, позволяя пользователю думать о результатах работы, а не о том, как их достичь. Этот набор примитивов был реализован на уровне абстракции ресурсов Puppet (Puppet's Resource Abstraction Layer).

Ресурсы Puppet должны быть уникальными для заданного узла. Вы можете иметь в распоряжении только один пакет с именем "ssh", одну службу с именем "sshd" и один файл с именем "/etc/ssh/sshd_config". Это ограничение предотвращает взаимные конфликты между различными частями ваших конфигураций и вы узнаете об этих конфликтах на раннем этапе процесса конфигурации. Мы ссылаемся на эти ресурсы по их типам и именам, т.е., `Package[ssh]` и `Service:sshd`. Вы можете использовать пакет и службу с одним и тем же именем, так как они относятся к различным типам, но не два пакета или службы с одним и тем же именем.

Второй ключевой инновацией в Puppet является возможность прямого указания зависимостей между ресурсами. Ранее используемые инструменты ставили своей целью выполнение индивидуальных задач без рассмотрения взаимосвязей между этими задачами; Puppet был первым инструментом, который явно устанавливал то, что зависимости являются первостепенной частью ваших конфигураций, которые, в свою очередь, должны моделироваться соответствующим образом. Он создавал граф ресурсов и их зависимостей в качестве одного из основных типов данных и практически все действия Puppet зависели от этого графа (называемого каталогом (*Catalog*)), его вершин и ребер.

Последним важным компонентом Puppet является язык конфигурации. Этот язык является декларативным и предназначен в большей степени для описания конфигурации, чем для полноценного программирования - он практически полностью повторяет формат конфигурации Nagios, но также был создан под значительным влиянием языков из состава CFEngine и Ruby.

В основе функциональных компонентов Puppet лежат два направляющих его развитие принципа: он должен быть настолько простым в использовании, насколько это возможно, причем предпочтение должно отдаваться удобству в использовании, а не возможностям; а также он должен разрабатываться в первую очередь в форме фреймворка и во вторую - приложения, таким образом при желании сторонние разработчики получат возможность создавать свои приложения на основе программных компонентов Puppet. Было понятно, что в дополнение к фреймворку необходимо также широко применяемое качественное приложение, но в первую очередь разработчики все равно занимались фреймворком, а не приложением. Многие люди все еще считают, что Puppet является этим самым приложением, а не фреймворком, на основе которого оно реализовано.

После создания первого прототипа Puppet, Luke стал в целом неплохим Perl-разработчиком с небольшим опытом разработки сценариев оболочки и небольшим опытом работы с языком C, большей частью полученным при работе с системой CFEngine. В дополнение к этому он имел опыт создания систем разбора данных для простых языков, который был получен при разработке двух таких систем для работы в составе небольших инструментов, а также повторной разработки с нуля системы разбора данных для CFEngine с целью упрощения ее поддержки (этот код не был передан проекту из-за небольших несовместимостей).

Решение об использовании динамического языка для реализации Puppet было принято достаточно быстро ввиду значительно более высокой производительности труда разработчика и распространения данного типа языков, но выбор самого оказался достаточно сложным. Начальные прототипы на языке Perl были отвергнуты, поэтому проводились эксперименты для поиска других языков. Была предпринята попытка использования языка Python, но Luke посчитал это язык значительно противоречащим его взгляду на мир. Услышав рассказ друга о преимуществах нового языка, Luke попробовал использовать язык Ruby и за четыре часа создал работающий прототип. В момент, когда началась полномасштабная разработка Puppet, язык Ruby был практически не известен, поэтому решение о его использовании было сопряжено с большим риском, но в этом случае производительность труда разработчика снова сыграла решающую роль в выборе языка. Главной отличительной чертой языка Ruby, по крайней мере от Perl, была простота создания неиерархических отношений классов, при этом язык не противоречил мыслительной деятельности разработчика Luke, что было критично.

18.2. Обзор архитектуры

Этот раздел в первую очередь посвящен описанию архитектуры реализации Puppet (т.е., описанию кода, который мы использовали для выполнения инструментом Puppet возложенных на него задач), но стоит также кратко обсудить архитектуру самого приложения (т.е., принцип взаимодействия его отдельных частей), ведь способ реализации приложения очень важен.

Инструмент Puppet был разработан для выполнения задач в двух режимах: в клиент/серверном режиме с центральным сервером и агентами, выполняющимися на отдельных узлах и в режиме без использования сервера, в котором отдельный процесс выполняет всю работу. Для достижения совместимости между этими режимами в рамках Puppet всегда использовался принцип внутренней сетевой прозрачности, поэтому при работе в двух режимах использовались одни и те же пути исполнения кода вне зависимости от того, осуществлялось ли взаимодействие посредством сети или нет. Для каждого исполняемого файла может быть установлен подходящий режим локального или удаленного доступа, но во всем остальном они будут вести себя идентично. Также следует отметить то, что вы можете использовать режим без сервера аналогично клиент-серверной конфигурации путем отправки файлов каждому клиенту и их последующего непосредственного разбора. В

в этом разделе будет описан клиент-серверный режим работы, так как он более прост для понимания ввиду рассмотрения отдельных компонентов, но следует помнить и о том, что информация из этого раздела также справедлива в случае работы в режиме без использования сервера.

Одно из определяющих архитектурных решений в рамках архитектуры приложения Puppet заключается в том, что клиенты не должны иметь доступ к модулям Puppet; вместо этого они должны получать спецификацию конфигурации, скомпилированную специально для них. Данный подход имеет множество достоинств: во-первых, вы будете следовать принципу снижения привилегий, в соответствии с которым каждый узел располагает только той информацией, которая предназначена для него (как он должен быть сконфигурирован), но не имеет доступа к информации о конфигурации других серверов. Во-вторых, вы можете полностью разделить операции и требования прав, необходимые для компиляции спецификации конфигурации (для этого может потребоваться доступ к централизованным хранилищам данных) и применения этой конфигурации. В-третьих, вы можете использовать отсоединенные узлы, на которых будет поддерживаться постоянная конфигурация без взаимодействия с центральным сервером, значит, конфигурация ваших серверов будет соответствовать спецификации даже в том случае, когда сервер прекратит работу и клиент отсоединится (как это происходит в случае мобильной установки или работы клиентов в "демилитаризованной зоне" сети (DMZ)).

Учитывая возможность этого выбора, процесс работы становится достаточно прямолинейным:

1. Процесс агента Puppet собирает информацию об узле, на котором он выполняется, и отправляет ее серверу.
2. Система разбора данных использует эту системную информацию и располагающиеся на локальном диске модули Puppet для компиляции спецификации конфигурации для этого определенного узла, после чего возвращает ее агенту.
3. Агент применяет полученную спецификацию конфигурации локально, таким образом изменяя локальное состояние узла, и заполняет отчет о результатах при содействии сервера.

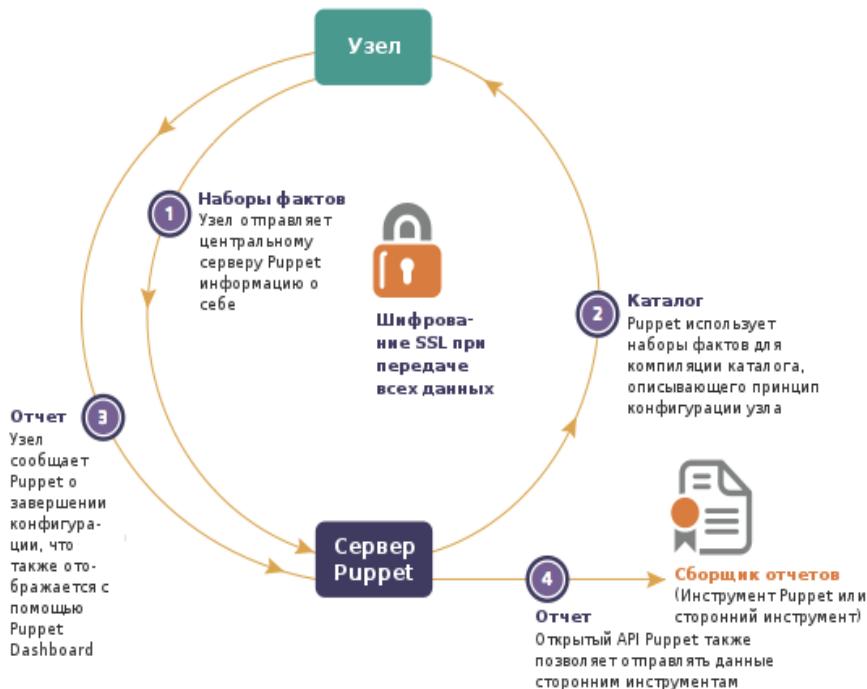


Рисунок 18.1: Потоки данных в Puppet

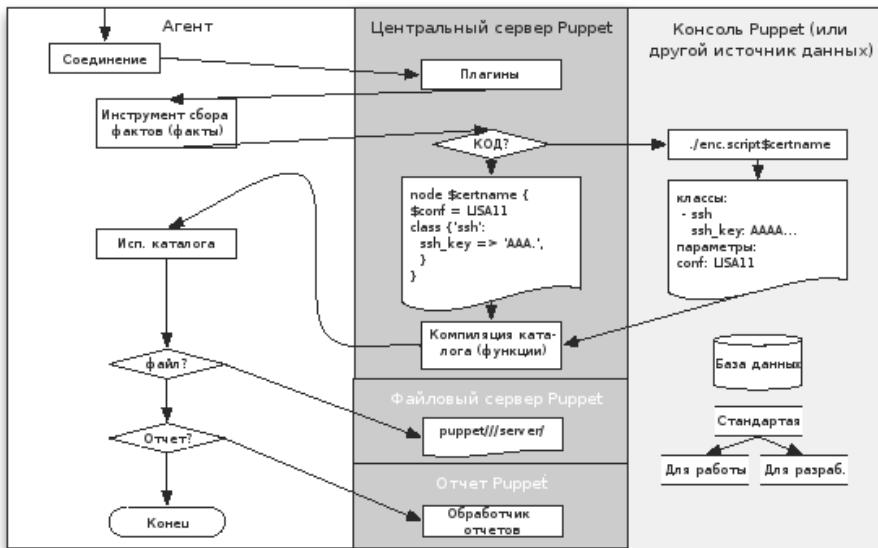


Рисунок 18.2: Управление потоком данных между процессами и компонентами Puppet

Таким образом, агент имеет доступ к информации о своей системе, ее конфигурации и каждому отчету, который он генерирует. Сервер располагает копиями всех этих данных и к тому же имеет доступ ко всем модулям Puppet и любым базам данных и службам, которые могут потребоваться для компиляции спецификации конфигурации.

Помимо всех компонентов, которые участвуют в формировании описанного потока данных и о которых мы поговорим позже, существует множество типов данных, используемых Puppet в ходе выполнения операций внутреннего взаимодействия. Эти типы данных являются критичными, так как с их помощью осуществляются все взаимодействия и они являются публично раскрываемыми типами данных, которые могут принимать или генерировать любые другие инструменты.

Наиболее важными типами данных являются:

- **Наборы фактов (Facts):** Системные данные, собираемые на каждой машине и используемые для компиляции спецификаций конфигурации.
- **Манифест (Manifest):** Файлы, содержащие код Puppet и обычно объединяемые в рамках коллекций, называемых "модулями".
- **Каталог (Catalog):** Граф ресурсов заданного узла для управления и установления зависимостей между ними.
- **Отчет (Report):** Набор всех событий, генерируемых во время использования заданного каталога.

Помимо наборов фактов, манифестов, каталогов и отчетов, Puppet поддерживает типы данных для работы с файлами, сертификатами (которые он использует для аутентификации), а также некоторые другие.

18.3. Анализ компонентов

Агент (Agent)

Первым компонентом, с которым вы можете столкнуться при запуске Puppet, является процесс `agent`. Традиционно он начинал работу после запуска отдельного исполняемого файла с именем `puppetd`, но в версии 2.6 мы приняли решение о сокращении количества исполняемых файлов до одного, поэтому на данный момент он может быть запущен с помощью команды `puppet agent`, по аналогии с тем, как работает `Git`. Сам по себе агент реализует небольшое количество функций; в первую очередь это функции управления конфигурацией и код, реализующий описанные выше аспекты работы на стороне клиента.

Инструмент сбора фактов (Facter)

Следующим после агента программным компонентом является внешнее приложение, называемое инструментом сбора фактов, который является по сути очень простым приложением для сбора информации об узле, на котором работает. Собираются такие данные, как название операционной системы, IP-адрес узла и имя узла, при этом функции инструмента сбора фактов достаточно просто расширяются, поэтому множество организаций добавляет свои собственные плагины для получения дополнительных данных. Агент отправляет данные, собранные инструментом сбора фактов, серверу, после чего последний вступает в рабочий процесс.

Внешний классификатор узла (External Node Classifier)

Первым компонентом, с которым мы столкнемся на стороне сервера, является внешний классификатор узла или ENC. Этот компонент принимает имя узла и возвращает простую структуру данных, содержащую высокоуровневую спецификацию конфигурации для данного узла. Внешний классификатор узла обычно является отдельной службой или приложением: это сделано для взаимодействия с другим проектом с открытым исходным кодом, таким, как Puppet Dashboard или Foreman, или для интеграции с существующими хранилищами данных, такими, как LDAP. Целью этого компонента является установление того, к каким функциональным классам принадлежит заданный узел и того, какие параметры должны быть использованы для конфигурации этих классов. Например, заданный узел может принадлежать классам `debian` и `webserver` и иметь параметр `datacenter` со значением `atlanta`.

Следует отметить, что для версии 2.7 Puppet компонент ENC не является обязательным; вместо его использования пользователи могут непосредственно описать конфигурации узлов с помощью кода Puppet. Поддержка компонента ENC была добавлена примерно через 2 года после выпуска первого релиза Puppet, так как мы поняли, что процесс классификации узлов фундаментально отличается от процесса их конфигурации, что делает более осмысленным разделение инструментов для решения этих задач, нежели расширение функций языка для поддержки возможности решения обоих задач. ENC всегда является рекомендуемым компонентом и на некотором этапе развития проекта станет необходимым (в тот момент, когда в составе Puppet будет поставляться достаточно удобный для использования компонент и это требование не будет усложнять работу).

После того, как сервер получает классификационную информацию от внешнего классификатора узла и системную информацию от инструмента сбора фактов (посредством агента), он добавляет эту информацию в объект `Node` и передает его компилятору.

Компилятор (Compiler)

Как упоминалось ранее, в составе Puppet реализован специальный язык для описания конфигураций систем. В реальности компилятор этого языка состоит из трех частей: похожая на Yacc система разбора, генерации и лексического анализа кода; группа классов, используемая для построения нашего дерева абстрактного синтаксического анализа (Abstract Syntax Tree - AST); и класс компилятора `Compiler`, который управляет взаимодействием всех этих классов, а также представляет API для этой части системы.

Наиболее сложный аспект работы компилятора заключается в том, что большая часть кода конфигурации Puppet загружается по необходимости после первого обращения (для сокращения времени загрузки и пресечения возможности появления не соответствующих действительности, касающихся отсутствующих и не нужных зависимостей записей в журнале), что подразумевает отсутствие явных вызовов, позволяющих загрузить код и произвести его разбор.

Система разбора кода Puppet использует обычный, [похожий на Yacc генератор систем разбора кода](#), созданный с использованием инструмента с открытым исходным кодом [Racc](#). К сожалению, в

момент начала разработки проекта Puppet не существовало генераторов лексических анализаторов с открытым исходным кодом, поэтому используется лексический анализатор собственной разработки.

Так как мы используем в Puppet дерево абстрактного синтаксического анализа, каждое объявление из набора грамматических конструкций Puppet преобразуется в экземпляр класса дерева синтаксического анализа (т.е., `Puppet::Parser::AST::Statement`) вместо непосредственного выполнения соответствующего действия и эти экземпляры классов AST компонуются в форме дерева по мере сокращения дерева грамматических конструкций. Это дерево абстрактного синтаксического анализа позволяет улучшить производительность в том случае, если на единственном сервере будут компилироваться конфигурации для многих сторонних узлов, так как становятся возможными однократный разбор кода и многократная компиляция. Также у нас появляется возможность выполнить обзор внутренних структур дерева абстрактного синтаксического анализа, что позволяет получить информацию и возможности, которыми мы не обладали бы в случае непосредственного разбора кода без преобразований.

В начале развития проекта Puppet было доступно только несколько подходящих примеров методов построения дерева абстрактного синтаксического анализа, поэтому используемый метод прошел множество этапов эволюционного развития и в итоге мы, по-видимому, пришли к его относительно уникальной версии. Вместо создания одного дерева абстрактного синтаксического анализа для всей конфигурации, мы создаем множество небольших деревьев, доступ к которым осуществляется на основе имен. Например, этот код:

```
class ssh {
    package { ssh: ensure => present }
}
```

создает новое дерево абстрактного синтаксического анализа, содержащее один экземпляр класса `Puppet::Parser::AST::Resource` и сохраняет это дерево под именем "ssh" в хэш-таблице для всех классов этого определенного окружения. (Я не буду рассматривать подробности реализации других связанных с классами конструкций, так как эта информация не является обязательной для продолжения данного описания).

При наличии дерева абстрактного синтаксического анализа и объекта описания узла `Node` (полученного от внешнего классификатора узла), компилятор может выбрать классы, описанные в рамках данного объекта (конечно же, если таковые имеются), найти и обработать их. В ходе этой обработки компилятор занимается построением дерева пространств действия переменных; каждый класс получает свое собственное пространство действия переменных, которое объединяется с пространством создающего его класса. Основной принцип создания динамических пространств действия переменных в рамках Puppet: если один класс включает в себя другой класс, то включенный класс может напрямую работать с переменными включившего его в свой состав класса. Такое поведение всегда было кошмаром для разработчиков, поэтому мы прорабатывали пути избавления от этой возможности.

Дерево пространств действия переменных (Scope tree) является временным и уничтожается после завершения компиляции, но в ходе компиляции также постепенно формируется один артефакт. Мы называем этот артефакт каталогом (Catalog) и он является всего лишь графом, представляющим ресурсы и их взаимодействия. В этом каталоге не сохраняются никакие описания переменных, управляющих структур или вызовов функций; все, что там хранится - это необработанные данные, которые могут быть достаточно просто сконвертированы в форматы JSON, YAML или в любые другие.

Во время компиляции мы формируем данные об отношении ресурсов к соответствующим классам; класс "содержит" все ресурсы, которые используются этим классом (т.е., описанный выше

пакет ssh содержится в классе ssh). Класс может содержать описание, которое само содержит либо дополнительные описания, либо отдельные ресурсы. Каталог по большей части является очень горизонтальным несвязанным графом: он содержит множество классов, глубина каждого из которых обычно не превышает нескольких уровней.

Одной из причиняющих неудобства особенностей данного графа является то, что он также содержит отношения "зависимостей", представленных такими данными, как описания службы требующей пакет (может быть, это сделано из-за того, что при установке пакета на самом деле создается служба), но эти отношения зависимостей фактически указываются в форме значений параметров ресурсов вместо вершин в структуре графа. Наш класс графа (с именем SimpleGraph, присвоенным по историческим причинам) не предоставляет возможности создания вершин для содержащихся ресурсов и зависимостей в рамках одного графа, поэтому нам приходится производить преобразования между ними с различными целями.

Транзакция (Transaction)

После того, как формирование каталога завершено (будем считать, что ошибки не произошли), он будет передан с помощью транзакции. В системе с разделенными клиентом и сервером, транзакция выполняется на стороне клиента, который принимает каталог, используя для этого протокол HTTP, как показано на Рисунке 18.2.

Класс транзакции системы Puppet является фреймворком для фактического вмешательства в работу системы, в то время, как все описанные нами ранее компоненты просто участвуют в процессе формирования и передачи объектов. В отличие от транзакций, которые выполняются в таких свойственных для них системах, как базы данных, транзакции системы Puppet не обладают такими свойствами, как атомарность.

Транзакция используется для выполнения относительно простой задачи: эта задача заключается в обходе графа, фактически представляющего различные взаимоотношения компонентов и проверке того, что состояние каждого из ресурсов было синхронизировано. Как было сказано выше, в ходе транзакции приходится преобразовывать вершины графа содержащихся ресурсов (т.е., вершины, указывающие, например, на то, что класс Class[ssh] содержит пакет Package[ssh] и службу Service[sshd]) в вершины графа зависимостей (т.е., в вершины, указывающие на то, что служба Service[sshd] зависит от пакета Package[ssh]), после чего выполняется стандартная топологическая сортировка графа с выбором каждого из ресурсов по очереди.

В отношении заданного ресурса мы осуществляем простой процесс обработки, состоящий из трех шагов: получение текущего состояния этого ресурса, сравнение его с желаемым состоянием и осуществление любых изменений, необходимых для устранения расхождений. Например, при использовании следующего кода:

```
file { '/etc/motd':
  ensure => file,
  content => "Welcome to the machine",
  mode   => 644
}
```

в ходе транзакции производится проверка содержимого и прав доступа к файлу /etc/motd, а в том случае, если они не совпадают с указанными, производятся изменения либо одного, либо обоих параметров. Если в файловой системе по пути /etc/motd каким-либо образом оказалась директория, будет сделана резервная копия всех файлов этой директории, после чего она будет удалена и заменена на файл с соответствующим содержимым и правами доступа.

Этот процесс изменения состояния системы фактически производится средствами простого класса `ResourceHarness`, в рамках которого полностью описан интерфейс между классами транзакции `Transcation` и ресурса `Resource`. Данный подход позволяет снизить количество соединений между классами и упрощает внесение изменений в самостоятельные классы.

Уровень абстракции ресурсов (Resource Abstraction Layer)

Класс транзакции является ключевым механизмом, участвующим в выполнении основной работы системы `Puppet`, но на самом деле вся работа выполняется на уровне абстракции ресурсов, который также является наиболее интересным компонентом `Puppet` с точки зрения его архитектуры.

Уровень абстракции ресурсов был первым компонентом, созданным в рамках проекта `Puppet`, и, в отличие от языка, он точно описывает то, что может сделать пользователь. Задачей этого компонента является определение назначения ресурса и способа выполнения работы в рамках системы с использованием ресурсов, а язык `Puppet` специально создан для указания ресурсов с использованием модели, понятной уровню абстракции ресурсов. Поэтому это также наиболее важный компонент системы, который сложнее всего изменить. Существует огромное количество вещей, которые нам хотелось бы изменить в рамках уровня абстракции ресурсов и мы уже реализовали множество критических улучшений данного компонента в течение многих лет (наиболее важным из них было добавление классов `Providers`), но, несмотря на это, большой объем работы в рамках этого компонента придется выполнить в будущем.

На уровне подсистемы компилятора мы создаем модели ресурсов и их типов с использованием отдельных классов (названных соответственно `Puppet::Resource` и `Puppet::Resource::Type`). Наша цель состоит в том, чтобы использовать эти классы также на уровне абстракции ресурсов, но на сегодняшний день модели этих двух элементов (ресурс и тип) создаются с использованием единственного класса `Puppet::Type`. (Класс назван некорректно из-за того, что он был создан задолго до того, как мы начали использовать термин "ресурс", в то время, когда мы использовали непосредственную сериализацию структур из памяти для осуществления взаимодействия между узлами, поэтому было достаточно сложно изменить имена классов.)

Во время создания класса `Puppet::Type` казалось разумным размещение данных ресурса и его типа в едином классе; кроме того, ресурсы являются всего лишь экземплярами типов ресурсов. Со временем, однако, стало понятно, что отношение между ресурсом и его типом не достаточно хорошо смоделировано в понятиях традиционной структуры наследования. Например, типы ресурсов описывают то, какие параметры может иметь ресурс, а не то, принимает ли он параметры (они все их принимают). Следовательно, наш базовый класс `Puppet::Type` реализует на уровне классов поведение, призванное установить поведение типов ресурсов, а также поведение на уровне экземпляров классов, направленное на установление их поведения. Также его задачей является управление регистрацией и получением типов ресурсов; если вам нужен тип "user", вы можете осуществить вызов `Puppet::Type.type(:user)`.

Это смешение типов поведения значительно затрудняет поддержку кода класса `Puppet::Type`. Весь класс состоит менее чем из 2,000 строк кода, но функционирует на трех уровнях - ресурсов, типов ресурсов и управления типами ресурсов, что делает его очень запутанным. Становится абсолютно понятно, из-за чего он является главной целью рефакторинга, но с помощью него не производится взаимодействия с пользователем, поэтому обычно сложно выделить ресурсы на его рефакторинг вместо непосредственной реализации новых возможностей.

Уровнем ниже класса `Puppet::Type` в рамках уровня абстракции ресурсов находятся классы двух основных типов, наиболее интересный из которых мы называем `Providers`. На начальном этапе разработки уровня абстракции ресурсов в каждом типе ресурса происходило смешение описания параметра с кодом, который реализовывал функции управления. Например, мы могли объявить параметр "content", после чего реализовать метод, с помощью которого можно будет прочитать

содержимое файла, а также другой метод, с помощью которого можно будет модифицировать содержимое этого файла:

```
Puppet::Type.newtype(:file) do
  ...
  newproperty(:content) do
    def retrieve
      File.read(@resource[:name])
    end
    def sync
      File.open(@resource[:name], "w") { |f| f.print @resource[:content] }
    end
  end
end
```

Этот пример значительно упрощен (в том смысле, что мы используем контрольные суммы при выполнении внутренних операций с файлами вместо строк с содержимым), но, несмотря на это, вы все равно получите необходимое представление о работе класса.

Использование необходимой модели управления ресурсами с учетом всего их многообразия, со временем стало невозможным. Проект Puppet на данный момент поддерживает 30 типов систем управления пакетами и было бы невозможно поддерживать все их средствами единственного типа ресурса Package. Вместо этого мы реализуем понятный интерфейс для описания типа ресурса. Предоставляющие свойства классы реализуют методы установки и получения значений для всех свойств типов ресурсов, названные очевидным образом. Например, ниже приведен образец класса, предоставляющего описанное свойство:

```
Puppet::Type.newtype(:file) do
  newproperty(:content)
end
Puppet::Type.type(:file).provide(:posix) do
  def content
    File.read(@resource[:name])
  end
  def content=(str)
    File.open(@resource[:name], "w") { |f| f.print(str) }
  end
end
```

При этом приходится затрагивать больший объем кода даже в простых случаях, но код гораздо проще понимать и поддерживать, особенно в том случае, когда возрастают либо количество свойств, либо количество классов, предоставляющих свойства.

В начале этого раздела я упоминал о том, что транзакция на самом деле не затрагивает систему напрямую, а вместо этого использует уровень абстракции ресурсов для взаимодействия с ней. Сейчас понятно, что предоставляющие свойства классы на самом деле выполняют необходимую работу. Фактически в общем случае только предоставляющие свойства классы реально вмешиваются в работу системы. Из транзакции поступает запрос содержимого файла и предоставляющий свойства класс считывает его; транзакция указывает на то, что содержимое файла должно быть изменено и предоставляющий свойства класс изменяет его. Следует отметить, однако, что предоставляющий свойства класс никогда самостоятельно не принимает решение о вмешательстве в работу системы - принятие решений происходит на уровне транзакции, после чего предоставляющие свойства классы выполняют работу. Это позволяет транзакции полностью контролировать систему без необходимости понимания аспектов работы с файлами, пользователями или пакетами и это разделение позволяет реализовать в рамках Puppet режим симуляции работы, при использовании которого можно с высокой вероятностью гарантировать то, что на систему не будет оказано воздействия.

Второй основной класс в рамках уровня абстракции ресурсов ответственен за сами параметры. Фактически мы поддерживаем три типа параметров: метапараметры, которые влияют на все типы ресурсов (т.е., любые ресурсы, которые вы можете использовать в режиме симуляции); параметры, являющиеся значениями, не копируемыми на диск (т.е., значениями, к примеру, указывающими на то, следует ли переходить по ссылкам в файлах); и свойства, при использовании которых моделируются аспекты поведения ресурса с изменением данных на диске (т.е., они могут отражать содержимое файла или состояние службы). Разница между свойствами и параметрами особенно сильно сбивает с толку людей, но если вы просто рассматриваете свойства как методы получения и установки значений в рамках классов, эта разница становится очевидной.

Создание отчетов (Reporting)

По мере обхода графа в ходе выполнения транзакции и использования уровня абстракции ресурсов для изменения системной конфигурации, происходит постепенное формирование отчета. Этот отчет в большей степени состоит из событий, генерируемых в ходе осуществления изменений в системе. События же, в свою очередь, предоставляют всестороннее отражение выполняемой работы: они содержат отметку времени, соответствующую времени изменения ресурса, предыдущее значение, новое значение, любое сгенерированное сообщение и отметку о том, успешным или безуспешным оказалось изменение (или о том, что активирован режим симуляции).

Эти события создаются в объекте `ResourceStatus`, который связан с каждым из ресурсов. Следовательно, для заданной транзакции у вас будет вся информация об использованных ресурсах и о любых произведенных изменениях наряду со всеми метаданными этих изменений, которые могут потребоваться вам.

По завершении транзакции происходит расчет и сохранение в отчете некоторых простейших характеристик, после чего отчет отправляется серверу (в случае соответствующей настройки). После отправки отчета процесс конфигурации считается завершенным и агент снова проходит в режим ожидания, либо его процесс просто завершается.

18.4. Инфраструктура

Сейчас, после того, как мы получили четкое представление о том, как и какие действия выполняет Puppet, стоит потратить немного времени на обсуждения компонентов кода, которые не выступают в роли отдельных систем, реализующих возможности, но при этом критичны для выполнения работы.

Плагины

Одной из замечательных характеристик Puppet является значительная гибкость. Существует по крайней мере 12 различных способов расширения возможностей Puppet, причем большинство этих способов предназначено для использования практически любым пользователем. Например, вы можете создать специальные плагины для работы в следующих областях:

- работа с типами ресурсов и специальными предоставляемыми свойствами классами
- работа в качестве обработчиков отчетов, предназначенных для таких целей, как хранение отчетов в специальной базе данных
- работа в качестве плагинов для фреймворка Inderector, предназначенных для взаимодействия с существующими хранилищами данных
- работа по формированию наборов фактов, с целью получения дополнительной информации о ваших узлах

Однако, распределенная природа Puppet обуславливает необходимость агентов в способе получения и загрузки новых плагинов. Следовательно, при каждом запуске Puppet в первую очередь загружаются все доступные серверу плагины. Эти плагины могут являться новыми типами ресурсов

или предоставляющими свойства классами, а также новыми фактами или даже новыми обработчиками отчетов.

Это позволяет значительно усовершенствовать агенты Puppet без изменения основных пакетов Puppet. Данная возможность особенно полезна при использовании специализированных установок Puppet.

Фреймворк Indirector

Вы наверняка уже поняли, что у нас существует традиция некорректного именования классов Puppet и, по мнению множества людей, этот случай заслуживает награды. Indirector является относительно стандартным фреймворком, построенным на основе принципа инверсии управления, со значительными возможностями расширения функций. Системы, созданные на основе принципа инверсии управления, позволяют отделить развертываемые функции от метода управления используемыми функциями. В случае Puppet это обстоятельство позволяет нам работать с множеством плагинов, предоставляющих различные функции, предназначенными для таких целей, как доступ к компилятору по протоколу HTTP или его загрузка в процессе работы, а также осуществлять переключение между ними при помощи небольших изменений конфигурации вместо вмешательства в код. Другими словами, фреймворк Indirector из состава Puppet является реализацией шаблона проектирования "service locator", описанного на странице Wikipedia с названием "Инверсия управления". Все взаимодействия одного класса с другим осуществляются посредством фреймворка Indirector с применением похожего на REST интерфейса (т.е., мы поддерживаем методы `find`, `search`, `save` и `destroy`), при этом переключение режима работы Puppet с бессерверного на клиент/серверный - в большей степени вопрос конфигурации агента таким образом, чтобы он использовал протокол HTTP в качестве конечной точки для получения каталога вместо конечной точки для доступа к компилятору.

Из-за того, что рассматриваемый фреймворк построен на принципе инверсии управления, в соответствии с которым конфигурация строго отделена от путей исполнения кода, этот класс может также сложно поддаваться отладке, особенно в том случае, если вы устанавливаете причину использования определенного пути исполнения кода.

Сетевые взаимодействия

Прототип Puppet был разработан летом 2004 года, когда главным вопросом относительно сетевого взаимодействия был вопрос о том, следует ли использовать XMLRPC или SOAP. Мы выбрали XMLRPC и этот протокол работал хорошо, но мы столкнулись с большей частью проблем, известных всем другим его пользователям: он не обуславливал использование стандартных интерфейсов для взаимодействия компонентов и в результате очень быстро приобрел излишнюю сложность. Мы также столкнулись со значительными проблемами в отношении использования памяти, так как кодирование данных для использования протокола XMLRPC подразумевало то, что каждый объект как минимум несколько раз размещался в памяти, что очень скоро привело к значительным затратам памяти при работе с файлами большого объема.

В рамках релиза 0.25 (работа над которым началась в 2008 году) мы начали процесс перевода всех сетевых взаимодействий на использование похожей на REST модели, но мы выбрали значительно более запутанный путь, чем простое изменение метода сетевого взаимодействия. Мы разработали фреймворк Indirector для использования в качестве стандартного фреймворка для межкомпонентного взаимодействия и сделали конечные точки REST одним из возможных вариантов осуществления этого взаимодействия. Реализация полной поддержки REST растянулась на два релиза и мы пока не до конца преобразовали код для использования формата JSON (вместо YAML) во всех операциях сериализации. Мы предприняли переход к использованию JSON по двум основным причинам: во-первых, обработка структур формата YAML средствами языка Ruby происходит очень медленно, при этом обработка структур формата JSON происходит значительно быстрее;

во-вторых, большая часть веб-приложений переходит на использование JSON и наблюдается тенденция создания более переносимых реализаций библиотек для работы с форматом JSON в отличие от формата YAML. Конечно же, в случае проекта Puppet первые варианты данных в формате YAML не были переносимы между языками, а также обычно не были переносимы между различными версиями Puppet, так как они в основном создавались в результате сериализации внутренних объектов Ruby.

В нашем следующем основном релизе Puppet мы наконец окончательно удалим код поддержки протокола XMLRPC.

18.5. Выученные уроки

Говоря о реализации, мы гордимся различными типами разделения, которые применены в Puppet: язык полностью отделен от уровня абстракции ресурсов, транзакция не может напрямую повлиять на систему и на уровне абстракции ресурсов не принимается самостоятельных рабочих решений. Это наделяет разработчика приложений значительными возможностями контроля над процессом работы приложения, а также открывает доступ к информации о том, что и почему происходит.

Потенциал расширения возможностей и конфигурации Puppet также очень важен, ведь любой человек может создавать приложения на основе Puppet без лишних сложностей и вмешательства во внутренние функции. Мы всегда реализуем возможности приложения с использованием тех же интерфейсов, которые мы рекомендуем использовать нашим пользователям.

Простота и легкость использования Puppet всегда были приоритетными направлениями развития. Проект все еще достаточно сложно развернуть и запустить, но этот процесс значительно проще тех, что необходимы для введения в строй других аналогичных инструментов. За простоту приходится расплачиваться инженерными решениями, особенно в форме поддержки и дополнительной работы, связанной с проектированием, но это стоит того, ведь важно позволить пользователям заниматься своими задачами вместо задач, связанных с используемым инструментом.

Возможности конфигурации Puppet по истине замечательны, но мы немного увлеклись их реализацией. Существует очень много способов объединения компонентов Puppet и очень просто создать на основе Puppet работающий инструмент, который не удовлетворит вас в итоге. Одной из основных долгосрочных целей является значительное сокращение количества параметров в рамках конфигурации Puppet для того, чтобы пользователь без сложностей не мог осуществить некорректную настройку и мы могли со временем упростить процесс обновления без беспокойства о нестандартных ситуациях.

Мы также очень медленно реализуем значительные изменения. Существуют важные рефакторинги, которые мы планировали провести в течение многих лет, но никогда не проводили. В итоге наши пользователи будут работать с более стабильной системой в краткосрочной перспективе, но при этом осложнится поддержка системы и внесение изменений в ее код.

Наконец, нам потребовалось слишком много времени для того, что бы понять, что наши цели, заключающиеся в упрощении системы, лучшим образом формулируются с использованием языка архитектуры. Как только мы начали говорить об архитектуре вместо абстрактного упрощения, мы обзавелись гораздо лучшим фреймворком для принятия решений о добавлении и удалении возможностей с лучшими способом взаимодействия для обоснования этих решений.

18.6. Заключение

Puppet является одновременно простой и сложной системой. Эта система состоит из множества работающих частей, но они достаточно слабо связаны друг с другом и каждая из них значительно

изменилась с момента начала развития проекта в 2005 году. Это фреймворк, который может использоваться для решения любых типов задач, связанных с конфигурацией, но при этом также простое и доступное приложение.

Наш будущий успех заключается в развитии простого и стабильного фреймворка и поддержании простоты использования приложения в ходе расширения его возможностей.

19. PyPy

Предполагается, что до прочтения этой главы вы ознакомились с некоторыми базовыми понятиями, относящимися к теории компиляторов и интерпретаторов, такими, как байткод и сворачивание констант.

19.1. Немного истории

Python является динамическим языком программирования высокого уровня. Он был создан голландским программистом Guido van Rossum в конце 1980 годов. Оригинальная реализация языка программирования от Guido представляет собой традиционный интерпретатор байткода, разработанный с использованием языка программирования C и впоследствии известный под именем CPython. На сегодняшний день существует также множество других реализаций Python. Среди наиболее известных реализаций можно выделить Jython, которая разработана с использованием языка программирования Java и позволяет осуществлять взаимодействие с кодом Java, IronPython, которая разработана с использованием языка программирования C# и позволяет взаимодействовать с фреймворком .Net от компании Microsoft, а также PyPy, которая будет рассматриваться в данной главе. CPython является все еще наиболее широко используемой реализацией и на сегодняшней день единственной реализацией, которая поддерживает синтаксические конструкции Python 3, следующего поколения языка программирования Python. В рамках данной главы будут описаны архитектурные решения, принятые в ходе разработки PyPy и отличающие эту реализацию от других реализаций языка Python и более того, от любых других реализаций динамических языков.

19.2. Обзор PyPy

При разработке PyPy использовался только язык программирования Python, за исключением немногочисленных заглушек на языке C. Дерево исходного кода проекта PyPy содержит два основных компонента: интерпретатор языка Python и набор инструментов для преобразования кода RPython. Интерпретатор языка Python является применяемым разработчиками окружением времени исполнения, используемым людьми при вызове реализации языка Python под названием PyPy. Фактически оно разработано с использованием подвида языка Python с именем Restricted Python (Python с ограничениями; обычно для его обозначения используется аббревиатура RPython). Цель разработки интерпретатора языка Python с использованием языка RPython заключается в реализации возможности подачи выходных данных интерпретатора на вход второй основной части PyPy, являющейся набором инструментов преобразования кода RPython. Инструмент преобразования кода RPython принимает код на языке RPython и преобразует его в код на выбранном языке более низкого уровня, наиболее часто этим языком является C. Это обстоятельство позволяет PyPy быть самодостаточной реализацией, что подразумевает использование для разработки того языка программирования, поддержка которого реализуется. Как мы увидим в данной главе, инструмент преобразования кода RPython также делает PyPy фреймворком для реализации динамических языков программирования общего назначения.

Мощные абстракции PyPy делают ее наиболее гибкой реализацией языка Python. Она поддерживает около 200 параметров конфигурации, которые позволяют осуществлять действия начиная с

выбора реализации сборщика мусора и заканчивая изменением параметров различных оптимизаций процесса преобразования кода.

19.3. Интерпретатор языка Python

Так как RPython является подвидом языка Python и строго соответствует его синтаксису, интерпретатор языка Python проекта PyPy может работать поверх другой реализации языка Python без преобразования кода. Конечно же, он будет работать чрезвычайно медленно, но при этом появляется возможность быстрого тестирования изменений в интерпретаторе. Также это обстоятельство позволяет использовать обычные инструменты отладки для языка Python для отладки интерпретатора. Большинство тестов интерпретатора PyPy может быть выполнено как при работе интерпретатора без преобразования кода, так и при работе интерпретатора с преобразованием кода. Это позволяет проводить быстрое тестирование в процессе разработки, а также проверять то, что интерпретатор при осуществлении преобразования кода ведет себя также, как и интерпретатор без преобразования кода.

По большей части детали реализации интерпретатора языка Python проекта PyPy схожи с деталями реализации интерпретатора CPython; интерпретаторы PyPy и CPython используют практически идентичные представления байткода и структуры данных в ходе интерпретации. Основным отличием между ними является то, что PyPy использует интересную абстракцию под названием "*пространства объектов*" ("object spaces" или сокращенно "objspaces"). Пространство объектов инкапсулирует данные, необходимые для представления и управления типами данных языка Python. Например, выполнение бинарной операции с двумя объектами Python или получение атрибута объекта в полной мере обрабатывается пространством объектов. Это обстоятельство позволяет освободить интерпретатор от необходимости получения любых деталей реализации объектов Python. Интерпретатор байткода рассматривает объекты Python как черные ящики и вызывает методы пространства объектов в любой момент, когда сталкивается с необходимостью осуществления манипуляций с ними. Например, ниже приведена простейшая реализация кода операции `BINARY_ADD`, который вызывается при комбинировании двух объектов с помощью оператора `+`. Обратите внимание на то, что операнды не проверяются интерпретатором; все действия по обработке объектов немедленно делегируются пространству объектов.

```
def BINARY_ADD(space, frame):
    object1 = frame.pop() # извлечение левого операнда из стека
    object2 = frame.pop() # извлечение правого операнда из стека
    result = space.add(object1, object2) # осуществление операции
    frame.push(result) # запись результата в стек
```

Абстракция пространства объектов имеет множество преимуществ. Она позволяет получать от объектов или передавать объектам новые реализации типов данных без модификации интерпретатора. Также ввиду того, что единственный способ осуществления манипуляций с объектами связан с использованием пространства объектов, на уровне пространства объектов могут осуществляться вмешательство, буферизация или запись операций с объектами. В ходе использования мощной абстракции пространств объектов в рамках PyPy были проведены эксперименты по внедрению технологии *переключения* (thunking), при использовании которой вычисление результатов может быть отложено, но осуществляться полностью прозрачно по требованию, а также технологии *создания исключений* (tainting), при использовании которой любая операция с объектом будет вызывать генерацию исключения (что полезно при передаче важных данных с использованием кода,зывающего недоверие). Наиболее важный способ применения пространства объектов, однако, будет обсуждаться в Разделе 19.4.

Пространство объектов, используемое в оригинальной версии интерпретатора PyPy, называется *стандартным пространством объектов* (standard objspace или std objspace для краткости). В дополнение к абстракции, предоставляемой системой пространства объектов, стандартное простран-

ство объектов предоставляет новый уровень абстракции: один и тот же тип данных может иметь множество реализаций. При его использования операции с типами данных осуществляются с применением мультиметодов. Это позволяет выбирать наиболее эффективное представление заданного набора данных. Например, тип long в рамках языка Python (очевидно, целочисленный тип данных большой разрядности) может быть представлен в виде стандартного целочисленного значения размером в одно машинное слово в том случае, если это значение достаточно мало. Более затратная в плане памяти и вычислений реализация типа данных произвольной точности большой разрядности должна использоваться только в случае необходимости. Существует даже реализация целочисленного типа данных Python на основе маркированных указателей. Контейнерные типы также могут быть специализированными по отношению к определенным типами данных. Например, в рамках PyPy реализован словарь (тип хэш-таблицы в Python), предназначенный для работы со строковыми ключами. Тот факт, что один и тот же тип данных может быть представлен различными реализациями, позволяет абсолютно прозрачно использовать объекты из кода уровня приложения; словарь для хранения строк идентичен словарю общего назначения и позволяет корректно отклонять нестроковые значения в случае их добавления.

В PyPy производится разделение между кодом уровня интерпретатора (interp-level) и кодом уровня приложения (app-level). Код уровня интерпретатора, который используется для реализации большей части интерпретатора, должен быть разработан с использованием языка RPython и подвергаться преобразованию. Он напрямую взаимодействует с пространством объектов и объектами Python в специальных обертках. Код уровня приложения всегда обрабатывается с помощью интерпретатора байткода PyPy. Ввиду простоты кода уровня интерпретатора на языке RPython по сравнению с кодом на языке C или Java, разработчики посчитали наиболее простым решением использование кода уровня приложения в некоторых частях интерпретатора. Следовательно, PyPy поддерживает встраивание кода уровня приложения в интерпретатор. Например, функции объявления `print` языка Python, с помощью которого производится запись данных объектов в поток стандартного вывода, реализованы на уровне приложения Python. Встроенные модули также могут быть разработаны с частичным использованием кода уровня интерпретатора и приложения.

19.4. Инструменты преобразования кода для языка RPython

Набор инструментов для языка RPython предназначен для осуществления нескольких фаз преобразования кода, в ходе которых код на языке RPython преобразовывается в код на целевом языке, обычно C. Высокоуровневое представление фаз преобразования кода показано на [Рисунке 19.1](#). Сами инструменты для преобразования кода разработаны с использованием языка Python (без ограничений) и тесно связаны с интерпретатором PyPy по причинам, которые будут освещены совсем скоро.

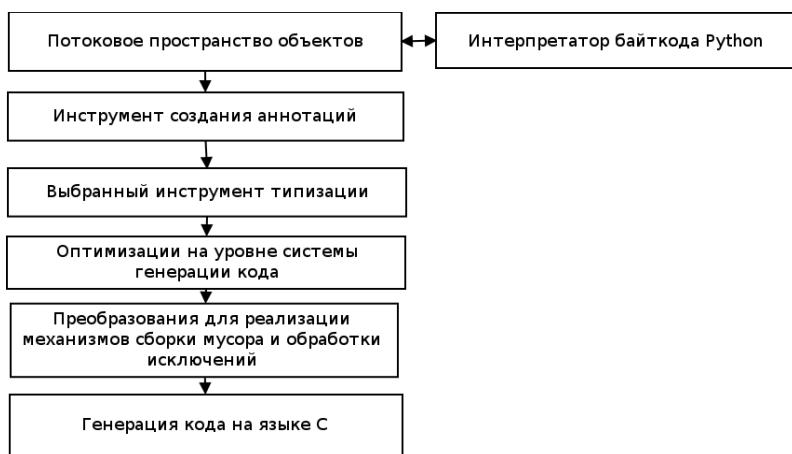


Рисунок 19.1: Шаги преобразования кода

Первой операцией, которую выполняет инструмент для преобразования кода, является загрузка программы на языке RPython в адресное пространство процесса. (Эта операция осуществляется с

использование стандартных для языка Python директив, предназначенных для поддержки операций загрузки модулей). Язык RPython налагает ряд ограничений на стандартные динамические функции языка Python. Например, функции не могут создаваться в процессе работы программы и одна и та же переменная не имеет возможности хранить значения несовместимых типов, примером которых могут служить целочисленное значение и объект. Несмотря на это, в момент начальной загрузки программы инструментом преобразования кода, она обрабатывается обычным интерпретатором языка Python и может использовать все динамические функции языка Python. Интерпретатор языка Python из состава PyPy является программой значительного размера, разработанной с использованием языка RPython, которая эксплуатирует эту возможность для реализации функций метапрограммирования. Например, она генерирует код для управления мультиметодами в стандартном пространстве объектов. Единственное требование заключается в том, что программа должна использовать корректные для языка RPython синтаксические конструкции перед началом новой фазы преобразования кода соответствующего инструмента.

Инструмент преобразования кода строит потоковые графы, отражающие программу на языке RPython, в ходе процесса с названием "*абстрактная интерпретация*" ("abstract interpretation"). В ходе абстрактной интерпретации происходит повторное использование интерпретатора языка Python из состава PyPy для интерпретации программ на языке RPython при наличии специального пространства объектов, называемого *потоковым пространством объектов* (flow objspace). Повторимся, что интерпретатор языка Python рассматривает объекты в программе как черные ящики, обращаясь к пространству объектов для выполнения любой операции. Потоковое пространство объектов вместо стандартного набора объектов языка Python работает только с двумя типами объектов: переменными и константами. Переменные представляют значения, не известные в момент преобразования кода, а константы, что не удивительно, представляют неизменные значения, которые известны в данный момент. Потоковое пространство объектов обладает базовыми возможностями сворачивания констант; если требуется выполнить операцию, все аргументы которой являются константами, в рамках него будет осуществлено статическое вычисление значения. Требования к неизменности значений и указания на необходимость использования констант в рамках языка RPython обозначаются значительно шире, чем в стандартном языке Python. Например, модули, которые несомненно изменяются в рамках языка Python, представляются константами в потоковом пространстве объектов, так как они не существуют в рамках языка RPython и должны быть свернуты в константы в потоковом пространстве объектов. По мере того, как интерпретатор языка Python интерпретирует байткод функций языка RPython, потоковое пространство объектов записывает операции, выполнение которых требуется. В рамках него осуществляется запись данных для всех конструкций условных ветвлений. Конечный результат абстрактной интерпретации функции является потоковым графом, состоящим из связанных блоков, причем каждый блок содержит одну или большее количество операций.

Прейдем к примеру процесса генерации потокового графа. Рассмотрим простую функцию вычисления факториала:

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

Потоковый график для функции выглядит следующим образом:

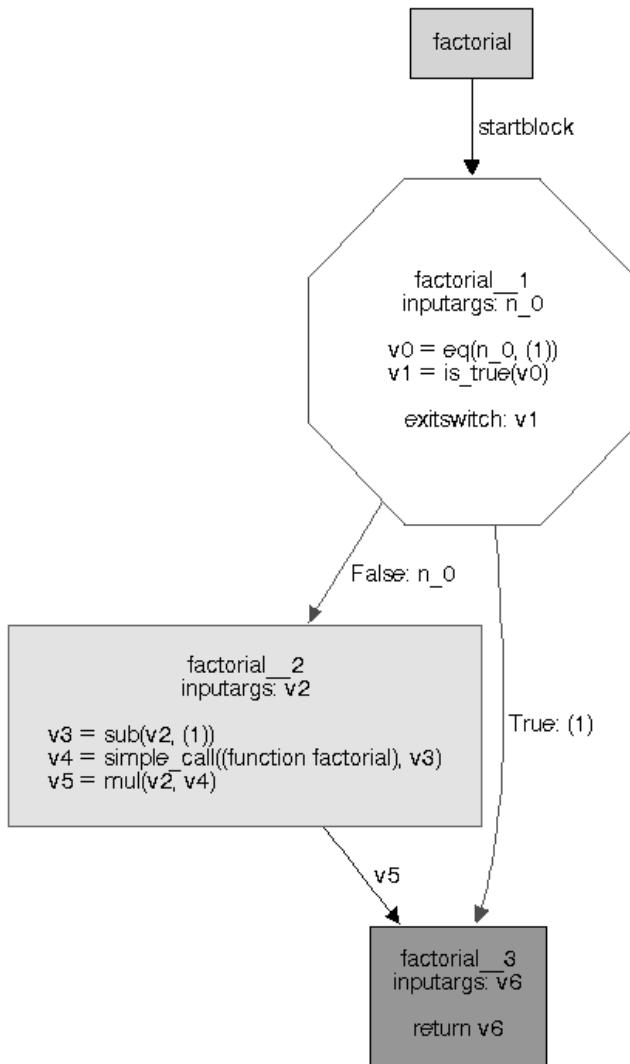


Рисунок 19.2: Потоковый график для функции вычисления факториала

Функция вычисления факториала была разделена на блоки, содержащие операции, записанные в рамках потокового пространства объектов. Каждый блок имеет входные аргументы и список операций с переменными и константами. Первый блок содержит выбор перехода в конце, который позволяет установить, к какому блоку перейдет управление после завершения выполнения первого блока. Решение о выходе может приниматься на основе значения какой-либо переменной или на основе того, было ли сгенерировано исключение в ходе выполнения последней операции блока. Передача управления осуществляется в соответствии с линиями между блоками.

Потоковый график, сгенерированный в потоковом пространстве объектов, представлен в статической единичной форме назначения (static single assignment form или SAA), являющейся промежуточном представлением, обычно используемым в компиляторах. Ключевая возможность представления SAA заключается в том, что значение каждой переменной присваивается единожды. Это свойство упрощает реализацию многих преобразований и оптимизаций, осуществляемых компиляторами.

После завершения генерации графа функции начинается фаза генерации аннотации. Инструмент создания аннотаций устанавливает типы результатов и аргументов для каждой из операций. Например, описанная выше функция вычисления факториала в аннотации будет принимать и возвращать целочисленные значения.

Следующая фаза называется типизацией (RTyping). В ходе типизации используется информация от инструмента создания аннотаций для того, чтобы преобразовать каждую из высокогорневых

операций, отображенных в рамках графа потоков данных, в низкоуровневые операции. Это первая часть процесса преобразования кода, для которой имеет значение выбранная система генерации кода. На основе системы генерации кода происходит выбор специфичного для программы инструмента типизации (RTyper). На данный момент инструмент типизации поддерживает две системы типов: низкоуровневую систему типов для систем генерации кода на таких языках, как C, а также высокоуровневую систему типов с классами. Высокоуровневые операции и типы языка Python приводятся в соответствие с выбранной системой типов. Например, для операции `add` с операндами, представленными целочисленными значениями в аннотации, будет сгенерирована операция `int_add` с низкоуровневой системой типов. Более сложные операции, такие, как поиск в хэш-таблицах, генерируют вызовы функций.

После типизации производятся некоторые оптимизации низкоуровневого потокового графа. Эти оптимизации по большей части являются такими стандартными применяемыми компиляторами оптимизациями, как сворачивание констант, удаление лишних операций резервирования памяти, а также удаление неиспользуемого кода.

Обычно код на языке Python периодически использует операции динамического резервирования памяти. Язык RPython, являясь производным языка Python, наследует этот шаблон проектирования, связанный с интенсивным резервированием памяти. Однако, в большинстве случаев эти резервирования памяти являются временными и локальными в рамках функций. *Удаление операций резервирования памяти (malloc removal)* является оптимизацией, призванной бороться с этими случаями. С помощью этой оптимизации производится удаление описанных операций резервирования памяти путем "разделения" ранее динамически созданного объекта на компоненты в том случае, если это возможно.

Для того, чтобы увидеть процесс удаления операций резервирования памяти в работе, предположим, что данная функция вычисляет Евклидово расстояние между двумя точками на поверхности простым способом:

```
def distance(x1, y1, x2, y2):
    p1 = (x1, y1)
    p2 = (x2, y2)
    return math.hypot(p1[0] - p2[0], p1[1] - p2[1])
```

После начальной типизации тело функции будет содержать следующие операции:

```
v60 = malloc((GcStruct tuple2))
v61 = setfield(v60, ('item0'), x1_1)
v62 = setfield(v60, ('item1'), y1_1)
v63 = malloc((GcStruct tuple2))
v64 = setfield(v63, ('item0'), x2_1)
v65 = setfield(v63, ('item1'), y2_1)
v66 = getfield(v60, ('item0'))
v67 = getfield(v63, ('item0'))
v68 = int_sub(v66, v67)
v69 = getfield(v60, ('item1'))
v70 = getfield(v63, ('item1'))
v71 = int_sub(v69, v70)
v72 = cast_int_to_float(v68)
v73 = cast_int_to_float(v71)
v74 = direct_call(math_hypot, v72, v73)
```

Этот код не является оптимальным по некоторым причинам. В функции резервируется память для хранения двух кортежей, которые никогда не покинут ее пределы. К тому же, без видимых причин используется косвенный способ доступа к полям кортежей.

После выполнения оптимизации, направленной на удаление операций резервирования памяти, будет получен следующий сжатый код:

```
v53 = int_sub(x1_0, x2_0)
v56 = int_sub(y1_0, y2_0)
v57 = cast_int_to_float(v53)
v58 = cast_int_to_float(v56)
v59 = direct_call(math_hypot, v57, v58)
```

Операции резервирования памяти для хранения кортежей были полностью исключены, а также были удалены операции косвенного доступа. Чуть позже мы увидим то, как аналогичная удалению операций резервирования памяти техника используется на уровне приложения Python для реализации JIT-компиляции в PyPy (Раздел 19.5).

PyPy также занимается созданием inline-функций. Как и в языках более низкого уровня, применение inline-функций позволяет улучшить производительность RPython. Как это не удивительно, использование таких функций также позволяет уменьшить размер итогового бинарного файла. Это происходит из-за того, что появляется возможность выполнения большего количества оптимизаций сворачивания переменных и удаления операций резервирования памяти, благодаря которым и уменьшается общий объем кода.

Программа, представленная на данный момент в форме оптимизированных низкоуровневых потоковых графов, передается системе генерации кода для непосредственной генерации исходного кода. Перед тем, как она сможет сгенерировать исходный код на языке C, с помощью специфической системы генерации кода для языка C должны быть осуществлены некоторые дополнительные преобразования. Одним из таких преобразований является преобразование исключений, при осуществлении которого код для обработки исключений преобразуется для использования техники неавтоматизированной раскрутки стека. Другой оптимизацией является вставка проверок глубины стека. С помощью этих проверок могут быть сгенерированы исключения в процессе выполнения программы в том случае, если выполняется слишком глубокая рекурсия. Места, в которых требуются проверки глубины стека, определяются путем подсчета циклов с использованием графа вызовов программы.

Другим преобразованием, выполняемым системой генерации кода для языка C, является добавление функций для сборки мусора (garbage collection - GC). RPython, как и Python, является языком, использующим механизм сборки мусора, при этом язык C не является таковым, поэтому реализация механизма сборки мусора должна быть добавлена. Для этого система преобразования кода, предназначенная для реализации механизма сборки мусора, преобразует потоковые графы программы в потоковые графы с возможностью сборки мусора. Системы преобразования кода сборки мусора из состава PyPy демонстрируют то, как в ходе преобразования можно абстрагироваться от незначительных деталей. В CPython, где используется подсчет ссылок на ресурсы, в рамках кода интерпретатора на языке C должно осуществляться тщательное отслеживание ссылок на объекты Python, с которыми производятся манипуляции. В ходе этого процесса в рамках всей кодовой базы осуществляется реализация схемы сборки мусора, но эта схема подвержена воздействию незначительных ошибок, которые может допускать человек. Система преобразования кода, предназначенная для реализации механизма сборки мусора в рамках PyPy решает обе эти проблемы; она позволяет бесшовно подключать и отключать различные схемы сборки мусора. Не так сложно использовать реализацию системы сборки мусора (одну из многих предоставляемых в рамках PyPy), просто изменив параметр конфигурации во время преобразования. Говоря о ошибках системы преобразования, следует упомянуть о том, что система преобразования кода, предназначенная для реализации механизма сборки мусора, также никогда не совершают ошибок в определении ссылок на ресурс и не забывает о необходимости информирования системы сборки мусора в момент, когда объект перестает использоваться. Мощь абстракции для сборки мусора заключается в том, что эта абстракция позволяет использовать реализации систем сборки мусора, которые практически не-

возможно реализовать вручную в рамках интерпретатора. Например, некоторые реализации систем сборки мусора из состава PyPy требуют наличия *барьера записи* (write barrier). Барьер записи является проверкой, которая должна производиться каждый раз, когда контролируемый системой сборки мусора объект помещается в другой контролируемый системой сборки мусора массив или структуру. Процесс установления барьера записи является трудоемким и может привести к ошибкам в случае ручной реализации, но он достаточно прост в том случае, когда выполняется автоматически системой преобразования кода, предназначенней для реализации механизма сборки мусора.

Наконец, система генерации кода получает возможность создания кода на языке C. Сгенерированный на основе низкоуровневых потоковых графов, код на языке C является уродливым нагромождением операторов `goto` и неочевидно названных переменных. Преимущество подхода, основанного на создании кода на языке C, заключается в том, что компилятор языка C может выполнить большую часть работы, заключающейся в сложных статических преобразованиях и требующейся для выполнения финальных оптимизаций циклов и резервирования регистров.

19.5. JIT-компиляция в PyPy

Python, как и большинство динамических языков программирования, традиционно отдает предпочтение гибкости в обмен на снижение производительности. Архитектура PyPy, обладая особенной гибкостью и широким спектром абстракций, затрудняет реализацию возможности очень быстрой интерпретации. Мощные абстракции пространств объектов и мульти методов в стандартном пространстве объектов не могут быть реализованы без последствий. В результате производительность не модифицированного интерпретатора PyPy будет в четыре раза ниже производительности интерпретатора CPython. Для того, чтобы не создавать репутацию медленного языка не только для нашей реализации, но и для языка Python в общем, в рамках PyPy был реализован динамический компилятор (*just-in-time compiler*, обычно обозначаемый с помощью абрэвиатуры JIT). С помощью JIT-компилятора часто используемые пути исполнения кода преобразуются в ассемблерное представление в процессе исполнения программы.

JIT-компилятор из состава PyPy использует преимущества уникальной архитектуры процесса преобразования кода в PyPy, описанной в Разделе 19.4. На самом деле PyPy не использует *Python-специфичный JIT-компилятор*; вместо него используется JIT-генератор. Генерация JIT-кода реализована просто в виде еще одной дополнительной фазы преобразования кода. Интерпретатор, желающий провести генерацию JIT-кода, должен осуществить два вызова специальных функций, называемых *указаниями jit* (*jit hints*).

JIT-генератор из состава PyPy является *трассирующим JIT-генератором* (*tracing JIT*). Это значит, что он определяет "горячие" (подразумевается часто используемые) циклы с целью их оптимизации путем компиляции в ассемблерный код. В момент, когда JIT-генератор принимает решение приступить к компиляции кода цикла, он записывает операции в рамках одной итерации цикла и этот процесс называется *трассировкой* (*tracing*). Эти операции впоследствии компилируются в машинный код.

Как было сказано ранее, JIT-генератору требуется только два указания от интерпретатора для генерации JIT-кода: `merge_point` и `can_enter_jit`. Функция `can_enter_jit` указывает JIT-генератору на начало цикла в рамках интерпретатора. При использовании интерпретатора языка Python это конец байткода `JUMP_ABSOLUTE`. (`JUMP_ABSOLUTE` заставляет интерпретатор перейти к началу цикла уровня приложения). Функция `merge_point` сообщает JIT-генератору о том, где он может безопасно вернуть управление интерпретатору. Это начало управляющего байткода цикла в интерпретаторе Python.

JIT-генератор вызывается после завершения фазы типизации RTyping в процессе преобразования кода. Повторим, что на данном этапе потоковые графы программы состоят из низкоуровневых

операций и практически готовы к участию в процессе генерации целевого кода. JIT-генератор обнаруживает описанные ранее указания от интерпретатора и заменяет их на вызовы, предназначенные для задействования скомпилированного JIT-кода в процессе работы приложения. После этого JIT-генератор записывает сериализованное представление потоковых графов для каждой функции, в рамках которой интерпретатор желает произвести JIT-оптимизацию. Эти сериализованные потоковые графы называются jit-кодами (jitcodes). Все функции интерпретатора в этот момент описываются с помощью низкоуровневых операций RPython. Jit-коды сохраняются в финальном бинарном файле для использования в процессе работы приложения.

В процессе работы приложения на уровне JIT-генератора поддерживается счетчик для каждого цикла, исполняемого в ходе работы программы. В момент, когда счетчик цикла преодолевает заданное в процессе конфигурации пороговое значение, осуществляется вызов JIT-генератора и начинается трассировка. Ключевым для процесса трассировки объектом является *мета-интерпретатор* (meta-interpreter). Мета-интерпретатор исполняет jit-коды, сформированные в процессе преобразования кода. Таким образом, происходит их интерпретация средствами основного интерпретатора, отсюда и название компонента. По мере трассировки цикла, он создает список выполняемых операций и записывает их в промежуточном представлении JIT (JIT intermediate representation - JIT IR), являющемся другим форматом записи операций. Этот список называется *трассировкой цикла* (trace of the loop). В моменты, когда мета-интерпретатор сталкивается с вызовом функции, преобразованной с использованием JIT-компиляции (функции, для которой существует jit-код), мета-интерпретатор входит в нее и записывает операции в оригинальную трассировку. Таким образом, процесс трассировки оказывает эффект, заключающийся в уменьшении глубины стека вызовов; единственным типом вызовов в рамках трассировки являются вызовы функций интерпретатора, которые выходят за пределы сферы действия jit-кода.

Мета-интерпретатор вынужден преобразовывать данные трассировки в свойства итерации цикла, трассировка которого производится. Например, в момент, когда мета-интерпретатор сталкивается с условным переходом в jit-коде, он, как и ожидается, должен выбрать один путь исполнения кода на основе состояния программы. При осуществлении выбора на основе информации, полученной во время исполнения программы, мета-интерпретатор записывает операцию в промежуточном представлении, называемую *охранной операцией* (guard). В случае условного перехода это будет операция `guard_true` или `guard_false` в отношении переменной условия. В большинстве арифметических операций также используются охранные операции, которые позволяют быть уверенными в том, что в ходе выполнения арифметической операции не произойдет переполнения. По существу охранные операции позволяют объявлять в коде предположения, которые делает мета-интерпретатор в ходе трассировки. В момент генерации ассемблерного кода охранные операции будут защищать ассемблерный код от выполнения в контексте, для работы в котором он не предназначен. Трассировка заканчивается в тот момент, когда мета-интерпретатор достигает той же операции `can_enter_jit`, с которой и началась трассировка. Теперь код промежуточного представления цикла может быть передан оптимизатору.

JIT-оптимизатор выполняет выполнить несколько классических оптимизаций компиляторов и множество оптимизаций, специфичных для динамических языков программирования. Наиболее важными оптимизациями, относящимися к последней категории, являются оптимизации *виртуальных* (virtuals) и *виртуализируемых* (virtualizables) объектов.

Виртуальные объекты являются объектами, о которых известно, то, что они не покидают пространство трассировки, что подразумевает тот факт, что они не передаются в качестве аргументов при вызове внешних функций, не подвергающихся преобразованию в jit-код. Структуры и массивы постоянной длины также могут быть виртуальными объектами. Для виртуальных объектов не должна резервироваться память и их данные могут храниться непосредственно в регистрах и в стеке. (Это очень напоминает случай удаления статических операций резервирования памяти, описанный в разделе об оптимизациях преобразованного кода). Оптимизация виртуальных объектов позволяет удалить неэффективные операции косвенной адресации и резервирования памяти в

рамках интерпретатора Python. Например, после преобразования в виртуальные объекты объектов Python для хранения в контейнерах целочисленных значений, эти значения могут быть извлечены из контейнеров и преобразованы в простые целочисленные значения длиной в машинное слово, после чего сохранены непосредственно в машинных регистрах.

Виртуализуемые объекты ведут себя в значительной степени аналогично виртуальным объектам, но могут покидать пространство трассировки (т.е., передаваться функциям, не преобразованным в jit-код). Объект фрейма интерпретатора Python, который содержит значения переменных и указатель инструкций, помечен как виртуализируемый. Это позволяет оптимизировать манипуляции со стеком и другие операции, выполняемые в рамках фрейма. Несмотря на то, что виртуальные и виртуализуемые объекты похожи, в плане реализации у них нет ничего общего. Виртуализуемые объекты обрабатываются в момент осуществления трассировки мета-интерпретатором. Это отличает их от виртуальных объектов, которые обрабатываются в процессе оптимизации. Причиной реализации такого подхода является то, что виртуализуемые объекты требуют особого обращения, так как они могут покидать пространство трассировки. В частности, мета-интерпретатор должен убедиться в том, что не преобразованные в jit-код функции, которые могут использовать виртуализуемые объекты, на самом деле не будут пытаться получить прямой доступ к данным их полей. Это требование выдвигается из-за того, что в jit-коде данные поля виртуализуемых объектов хранятся в стеке и регистрах, поэтому данные реального виртуализированного объекта могут устареть в сравнении с текущими значениями, используемыми в рамках jit-кода. В процессе генерации jit-кода, код, получающий доступ к виртуализируемому объекту, модифицируется для проверки того, исполняется ли ассемблерный код, полученный в результате jit-компиляции. В том случае, если он исполняется, в рамках jit-кода осуществляется запрос обновления полей объекта на основе данных, полученных при выполнении ассемблерного кода. Дополнительно в тот момент, когда происходит возврат управления из внешней функции в jit-код, управление передается интерпретатору.

После проведения оптимизации данные трассировки готовы для преобразования в ассемблерный код. Так как промежуточно представление кода JIT само по себе является достаточно низкоуровневым, генерация ассемблерного кода не является очень сложной задачей. Большинство операций из промежуточного представления соответствуют всего лишь нескольким ассемблерным операциям для архитектуры x86. Система резервирования регистров использует простой линейный алгоритм. На данный момент увеличение затрат времени для использования более оптимизированного алгоритма резервирования регистров в обмен на генерацию немного более качественного кода не оправдывает себя. Наиболее сложными аспектами генерации ассемблерного кода являются интеграция сборщика мусора и реализация механизма восстановления состояния охранных операций. Сборщик мусора должен отслеживать корневые элементы стека в рамках генерируемого jit-кода. Эта возможность реализуется путем специальной поддержки динамических карт корневых элементов стека в рамках сборщика мусора.

При неудачном выполнении охранной операции скомпилированный ассемблерный код более не считается корректным и управление должно быть передано интерпретатору байткода. Эта операция передачи управления является одной из наиболее сложных частей реализации механизма JIT-компиляции, так как данные состояния интерпретатора должны быть воссозданы на основе данных состояния регистров и стека в момент неудачного завершения охранной операции. Для каждой охранной операции в рамках ассемблерного кода создается компактное описание того, где находятся все значения для воссоздания данных состояния интерпретатора. В случае неудачного выполнения охранной операции начинается выполнение функции, которая декодирует это описание и передает данные для восстановления на более высокий уровень для осуществления операции восстановления данных состояния. Неудачно выполненная охранная операция может находиться по середине пути исполнения запутанного кода операции, поэтому интерпретатор не сможет просто начать исполнение следующего кода операции. Для решения этой проблемы PyPy использует интерпретатор "черной дыры" (blackhole interpreter). Интерпретатор "черной дыры" выполняет операции в рамках jit-кода с позиции неудачного выполнения охранной операции до достижения

следующей точки безопасной передачи управления. Он не записывает данные о любых операциях, которые выполняет. Процесс некорректного завершения охранной операции проиллюстрирован на [Рисунке 19.3](#).

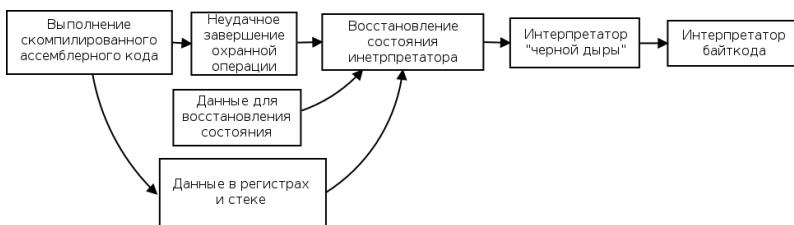


Рисунок 19.3: Передача управления интерпретатору при неудачном выполнении охранной операции

Как было описано до этого момента, JIT-компиляция будет по существу бесполезной для любого цикла с часто изменяющимися условиями, так как неудачное выполнение охранной операции предотвратит выполнение ассемблерного кода для большого количества итераций. Каждая охранная операция поддерживает счетчик неудачных выполнений. После того, как количество неудачных выполнений преодолевает определенное пороговое значение, JIT-генератор начинает трассировку с той точки, где произошло неудачное завершение охранной операции вместо передачи управления интерпретатору. Эта дополнительная трассировка называется мостом (bridge). В момент, когда трассировка достигает завершения цикла, мост оптимизируется и компилируется, после чего оригинальный бинарный код цикла обновляется для того, чтобы после выполнения охранной операции управление переходило к новому мосту вместо кода после неудачного выполнения. Таким образом, циклы с динамическими условиями могут быть преобразованы в jit-код.

Насколько же хорошо зарекомендовали себя техники JIT-компиляции, применяемые в PyPy? В момент работы над этой главой среднее геометрическое для значений производительности реализаций PyPy и CPython указывает на пятикратное преимущество в быстродействии первой при использовании полного набора тестов производительности. При использовании JIT-компиляции код Python уровня приложений имеет возможность более быстрой работы, чем код уровня интерпретатора. Разработчики PyPy недавно столкнулись с замечательной задачей, заключающейся в необходимости реализации циклов уровня интерпретатора в рамках кода уровня приложения Python для достижения лучшей производительности.

Наиболее важным является то, что система JIT-компиляции не является специфичной для языка Python и это означает, что она может быть применена при разработке любого интерпретатора на основе фреймворка PyPy. Это не обязательно должен быть интерпретатор языка программирования. Например, JIT-компиляция используется для работы механизма регулярных выражений языка Python. NumPy является мощным модулем для работы с массивами языка Python, используемым при выполнении численных операций и научных исследований. В PyPy имеется экспериментальная повторная реализация модуля NumPy. Она использует мощные механизмы JIT-компиляции PyPy для ускорения операций с массивами. Несмотря на то, что реализация модуля NumPy все еще находится на раннем этапе своего развития, начальные замеры производительности выглядят многообещающе.

19.6. Недостатки архитектуры

Хотя разработка приложений на языке RPython в любом случае проще, чем на языке C, опыт такой разработки может привести к разочарованию. В первое время сложно использовать применяющуюся в нем явную типизацию. Не все возможности языка программирования Python поддерживаются, а на поддерживаемые возможности накладываются произвольные ограничения. Язык RPython не имеет формальной спецификации и все принимаемые системой преобразования кода синтаксические конструкции могут меняться день ото дня из-за того, что язык RPython адаптируется для удовлетворения требований фреймворка PyPy. Автору этой главы часто удается создавать

программы, в ходе получасовой обработки которых с помощью системы преобразования кода выводится непонятное сообщение об ошибке и процесс преобразования прекращается.

Тот факт, что система преобразования кода RPython производит анализ всей программы, создает несколько практических проблем. Любое минимальное изменение преобразуемого кода приводит к необходимости повторного преобразования кода всего интерпретатора. На данный момент процесс преобразования кода растягивается на 40 минут при использовании быстрой современной системы. Эта задержка особенно раздражает тогда, когда тестируются изменения, затрагивающие систему JIT-компиляции, ведь для измерения производительности необходимо произвести преобразование кода интерпретатора. Требование наличия всего кода программы перед его преобразованием подразумевает то, что модули, содержащие код на языке RPython, не могут быть скомпилированы и загружены отдельно от основного кода интерпретатора.

Уровни абстракции в PyPy не всегда так четко разделены, как это выглядит в теории. Хотя технически JIT-генератор должен иметь возможность создания замечательного JIT-компилятора для языка с использованием только двух упомянутых ранее указаний, в реальности он работает лучше с одним определенным кодом, нежели с другим. Был проведен большой объем работы для того, чтобы интерпретатор Python был "лучше совместим с процессом генерации jit-кода", включая реализацию большего количества JIT-указаний и даже новые структуры данных, специально оптимизированные для работы с JIT-генератором.

Большое количество уровней абстракции PyPy может сделать поиск ошибок достаточно сложным процессом. Ошибка интерпретатора Python может находиться в самом коде интерпретатора, либо скрываться где-либо в семантиках языка RPython и инструментарии для преобразования кода. Отладка особенно осложняется в том случае, когда ошибка не может быть воспроизведена без преобразования кода интерпретатора. Обычно в этих случаях прибегают к использованию отладчика GDB по отношению к приложению, скомпилиированному на основе практически нечитаемого автоматически сгенерированного исходного кода на языке C.

Преобразование даже ограниченного подмножества языка Python в такой более низкоуровневый язык, как C, не является простой задачей. Фазы процесса преобразования, описанные в Разделе 19.4, на самом деле не являются независимыми. В процессе преобразования кода производится создание аннотаций и типизация функций, при этом система создания аннотаций располагает информацией о низкоуровневых типах. Следовательно, система преобразования кода для языка RPython сталкивается с запутанной схемой зависимостей. Система преобразования может немного упростить ее в некоторых местах, но этот процесс не прост и не доставляет удовольствия.

19.7. Немного о процессе разработки

В ходе борьбы со сложностью реализации (обратитесь к Разделу 19.6) в рамках фреймворка PyPy начали применяться несколько так называемых "гибких" методологий разработки. Наиболее важной из них является разработка через тестирование. Все новые возможности и исправления ошибок должны сопровождаться тестами, позволяющими установить корректность их реализации. Интерпретатор языка Python из состава фреймворка PyPy также проходит тестирование с использованием набора тестов для обнаружения регрессий проекта CPython. Система тестирования фреймворка PyPy `py.test` была выделена из состава проекта и на данный момент используется многими другими проектами. При разработке PyPy также применяется система непрерывной интеграции, с помощью которой выполняется набор тестов и осуществляется перенос проекта на множество платформ. Бинарные файлы для всех платформ создаются ежедневно, после чего они подвергаются тестированию с помощью набора тестов. Все эти тесты позволяют быть уверенным в том, что различные компоненты функционируют в штатном режиме вне зависимости от того, какие изменения сложной архитектуры проекта были осуществлены.

Существует строгая культура проведения экспериментов в рамках проекта PyPy. Разработчики должны создавать ветви кода в репозитории Mercurial. В рамках этих ветвей могут быть реализованы связанные с разработкой проекта идеи без дестабилизации основной ветви кода. Идеи не всегда успешно реализуются в рамках этих ветвей исходного кода, поэтому некоторые ветви остаются в заброшенном состоянии. Во всяком случае, разработчики проекта PyPy очень настойчивы. Наиболее известным доказательством этого утверждения является тот факт, что современная система JIT-компиляции PyPy появилась в ходе *пятой* попытки добавления функций JIT-компиляции в фреймворк PyPy!

Проект PyPy также известен своими инструментами визуализации. Визуализации графов потоков данных из Раздела 19.4 являются одним из примеров применения этих инструментов. В составе проекта PyPy есть также инструменты для демонстрации вызовов сборщика мусора с течением времени и обзора деревьев разбора регулярных выражений. Особенно интересным инструментом является jitviewer - программа, которая позволяет визуализировать уровни преобразованной в JIT-код функции при преобразовании из байткода Python в промежуточное представление JIT с последующим преобразованием в ассемблерный код. (Вывод программы jitviewer показан на [Рисунке 19.4](#).) Инструменты визуализации помогают разработчикам понимать принципы взаимодействия множества уровней абстракции фреймворка PyPy.

```

LOAD_FAST
    guard(i6 == 2)
    guard_NONNULL(p9, ConstClass(W_IntObject), descr=<Guard25>)
    guard(i4 == 0)
LOAD_CONST
    guard(p3 == ConstPtr(ptr15))
COMPARE_OP
    i16 = ((pypy.objspace.std.intobject.W_IntObject)p9).inst_intval [pure]
    i18 = i16 < 10000
    guard(i18 is true)
POP_JUMP_IF_FALSE
LOAD_FAST
LOAD_CONST
BINARY_MODULO
    i20 = i16 == -9223372036854775808
    guard(i20 is false)
    i22 = int_mod(i16, 2)
    i24 = int_rshift(i22, 63)
    i25 = 2 & i24
    i26 = i22 + i25
POP_JUMP_IF_FALSE
    i27 = int_is_true(i26)
    guard(i27 is false)
LOAD_FAST
    guard_NONNULL(p8, ConstClass(W_IntObject), descr=<Guard31>)
LOAD_CONST
INPLACE_ADD
    i30 = ((pypy.objspace.std.intobject.W_IntObject)p8).inst_intval [pure]
    i32 = int_add_ovf(i30, 1)
    guard_no_overflow(descr=<Guard32>)
STORE_FAST
LOAD_FAST

```

Рисунок 19.4: Программа jitviewer выводит байткод Python и соответствующие операции промежуточного представления JIT

19.8. Резюме

Интерпретатор языка Python рассматривает объекты как черные ящики и позволяет полностью описывать их поведение в рамках пространства объектов. Отдельные пространства объектов могут позволять объектам языка Python расширять свои возможности. Подход, заключающийся в использовании пространства объектов, позволяет использовать технику абстрактной интерпретации в процессе преобразования кода.

Система преобразования кода RPython позволяет реализовывать такие возможности, как сборка мусора и обработка исключений, абстрагированные от интерпретатора языка программирования.

Она также делает возможным использование фреймворка PyPy на множестве различных платформ при использовании различных систем генерации кода.

Одним из наиболее важных методов применения архитектуры преобразования кода является возможность использования JIT-генератора. Обобщенная архитектура JIT-генератора позволяет использовать возможности JIT-компиляции для новых языков программирования и таких подвидов языков, как регулярные выражения. PyPy является быстрейшей реализацией языка программирования Python благодаря использованию встроенного JIT-генератора.

Хотя большая часть усилий в ходе разработки фреймворка PyPy направлена на развитие интерпретатора языка Python, фреймворк PyPy может использоваться для реализации интерпретатора любого динамического языка программирования. В течение многих лет на основе фреймворка PyPy создавались незаконченные интерпретаторы для языков JavaScript, Prolog, Scheme и IO.

19.9. Выученные уроки

Наконец, перечислим несколько уроков, извлеченных из процесса разработки проекта PyPy:

Повторяемый рефакторинг кода является обычно необходимым процессом. Например, изначально было выдвинуто предложение обработки высокоуровневых потоковых графов с помощью системы генерации кода на языке C! Прошло несколько рефакторингов кода до того момента, как был реализован текущий многофазный процесс преобразования кода.

Наиболее важным уроком, извлеченным из процесса разработки PyPy, является понимание мощности абстракции. В PyPy абстракции служат для разделения областей реализаций. Например, возможность автоматической сборки мусора языка RPython позволяет разработчику, работающему с интерпретатором, не беспокоиться об управлении памятью. В то же время, абстракции имеют подсознательную значимость. Работа с цепочкой преобразования кода подразумевает жонглирование различными фазами преобразования на уровне воображения. Поиск уровня абстракции, на котором допущена ошибка может быть также затруднен из-за применения абстракций; нарушение абстракций, когда происходит подмена низкоуровневых компонентов, которые должны быть взаимозаменяемыми приводит к нарушению работы высокоуровневого кода, что является извечной проблемой. Важно использовать тесты для проверки того, что все части системы работоспособны, поэтому изменение в одной системе не приведет к неработоспособности другой. Говоря более конкретно, абстракции могут замедлить программу, создавая большое количество косвенных преобразований.

Гибкость языка (R)Python, используемого в качестве языка реализации, позволяет проводить эксперименты с новыми возможностями языка Python (или даже с новыми языками) достаточно просто. Из-за своей уникальной архитектуры проект PyPy будет играть важную роль в будущей реализации языка Python и других динамических языков программирования.

20. SQLAlchemy

Глава 20 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

SQLAlchemy является тулkitом для работы с базами данных, а также системой объектно-реляционного отображения (ORM) для языка программирования Python, впервые представленной общественности в 2005 году. С самого начала реализация данного программного продукта осуществлялась с целью предоставления законченной системы для работы с реляционными базами данных в Python, которая будет основываться на API для работы с базами данных языка Python (DBAPI). Даже возможности ранних релизов SQLAlchemy привлекли широкое внимание общественности. Среди ключевых возможностей системы следует выделить большую скорость работы с

сложными SQL-запросами и отображениями объектов, а также реализацию шаблона проектирования "рабочей единицы" ("unit of work"), которая позволяет создать хорошо автоматизированную систему для сохранения информации в базе данных.

Начав свое развитие с небольшой по объему ограниченной реализации концепции, система SQLAlchemy быстро продолжила развиваться, проходя множество этапов преобразований и переработок, переходя к использования все новых внутренних архитектурных решений и публичных API по мере продолжающегося роста пользовательской базы. Ко времени выпуска версии 0.5 в январе 2009 года код системы SQLAlchemy начал стабилизироваться и широко внедряться. По мере выпуска версий 0.6 (в апреле 2010 года) и 0.7 (в мае 2011 года) было продолжено внесение улучшений в области архитектуры и API для реализации настолько производительной и стабильной библиотеки, насколько это возможно. На момент написания этой главы SQLAlchemy используется в большом количестве организаций в различных областях применения и считается многими стандартом де-факто системы для работы с реляционными базами данных при использовании языка программирования Python.

20.1. Сложность создания слоя абстракции для баз данных

При использовании термина "слой абстракции для баз данных" принято считать, что имеется в виду система взаимодействия с базой данных, которая скрывает большую часть подробностей о том, как данные хранятся и извлекаются. Этот термин иногда трактуется более радикально и в этом случае считается, что система должна скрывать не только специфику используемой реляционной базы данных, но даже и подробности формирования самих реляционных структур, а также то, является или не является используемое хранилище данных реляционным.

Наиболее часто критики инструментов для объектно-реляционного отображения основывают свои утверждения на предположении о том, что главной целью подобного инструмента является "скрытие" факта использования реляционной базы данных, выполнение задачи по конструированию запросов и взаимодействию с базой данных и скрытие множества подробностей реализации этого взаимодействия. Главной характерной чертой данного скрытия является возможность перевода операций создания и осуществления запросов реляционных структур из ведения разработчика в ведение прозрачно работающей библиотеки.

Те, кто имеет богатый опыт работы с реляционными базами данных, знают о том, что этот подход является абсолютно не практичным. Реляционные структуры и SQL-запросы являются очень функциональными и входят в состав основных архитектурных элементов приложений. То, как эти структуры должны проектироваться, организовываться и обрабатываться при работе с запросами, зависит не только от желаемых данных, но также и от структуры информации. В том случае, если эти возможности будут скрываться, не будет большого смысла в использовании в первую очередь реляционной базы данных.

Проблема согласования приложений, которые пытаются скрывать реализацию используемой реляционной базы данных с фактом, заключающимся в том, что реляционные базы данных требуют специфического подхода, обычно называется "проблемой объектно-реляционного несоответствия". SQLAlchemy предлагает сравнительно новый подход к решению данной проблемы.

Подход к созданию слоя абстракции для баз данных, используемый в рамках SQLAlchemy

SQLAlchemy предполагает, что разработчик пожелает использовать реляционную форму для своих данных. Система, которая изначально устанавливает и скрывает схему и принятые для формирования запросов архитектурные решения, ухудшает свои пользовательские качества при работе с реляционными базами данных, что ведет к появлению всех классических проблем несоответствия.

В то же время, реализация этих решений может и должна быть произведена в соответствии с высокоуровневыми шаблонами проектирования настолько, насколько это возможно. Установление связи модели объектов со схемой и реализация этой связи с помощью запросов является скучным занятием. Использование инструментов для автоматизации этих задач позволяет сделать процесс разработки приложения более коротким, понятным и эффективным, а также позволяет завершить его в течение того же промежутка времени, который мог потребоваться для разработки позволяющего выполнить эту задачу кода вручную.

В этом случае SQLAlchemy рассматривается как тулkit, который позволяет повысить значение роли разработчика с пассивного пользователя решений, предлагаемых библиотекой, до архитектора/создателя реляционных структур и связей между этими структурами и приложением. Раскрывая реляционные концепции, SQLAlchemy использует идею "неполной абстракции", принуждая разработчика к созданию специального и в то же время полностью автоматизированного уровня взаимодействия между приложением и реляционной базой данных. Инновация SQLAlchemy заключается в степени, до которой эта система позволяет осуществлять автоматизацию, причем данная автоматизация не снижает степень контроля разработчика над реляционной базой данных.

20.2. Дихотомия между основными задачами и объектно-реляционным отображением

Основной задачей, поставленной при разработке SQLAlchemy, было предоставление тулкита, который раскрывал бы каждый уровень взаимодействия с базой данных в рамках богатого набора функций API, разделяя задачи на две основных категории, известные как основные (Core) и связанные с объектно-реляционным отображением (ORM). Основные задачи заключаются во взаимодействии с API для работы с базами данных языка Python (DBAPI), выводе текстовых выражений SQL, понятных базе данных, а также управлении схемами. Все эти возможности доступны через публичные API. ORM, или объектно-реляционное отображение реализуется в рамках специфической библиотеки, созданной для работы с основными функциями системы. Объектно-реляционное отображение, реализуемое в рамках SQLAlchemy, является одним из неограниченного количества возможных уровней абстракции объектов, которые могут быть реализованы с использованием основных функций, причем многие разработчики и организации создают свои приложения, непосредственно использующие основные функции системы.

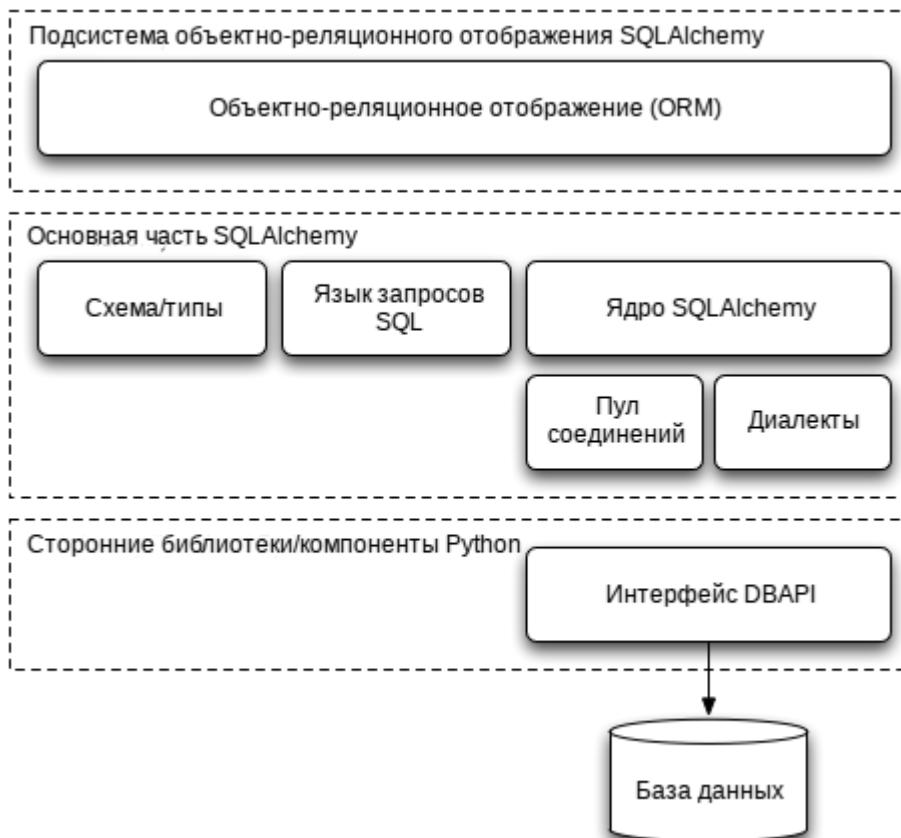


Рисунок 20.1: Диаграмма уровней SQLAlchemy

Разделение на основную часть и объектно-реляционное отображение всегда было наиболее характерной чертой SQLAlchemy, при этом данная черта имеет как достоинства, так и недостатки. Явно реализованная основная часть SQLAlchemy предполагает установление связи между атрибутами класса объектно-реляционного отображения и структурой, известной как `Table`, вместо непосредственной связи с названиями столбцов в строковом формате, как это установлено в базе данных; осуществление запроса `SELECT` с использованием структуры с именем `select` вместо объединения атрибутов объектов непосредственно с выражениями в строковой форме; а также прием результирующих строк с использованием фасадного класса с именем `ResultProxy`, который прозрачно отображает структуру `select` на каждую из результирующих строк вместо передачи данных непосредственно от курсора из базы данных в заданный пользователем объект.

Элементы основной части могут быть незаметны в очень простых приложениях, использующих объектно-реляционное отображение. Однако, из-за того, что основные функции тщательно интегрированы в код объектно-реляционного отображения для предоставления возможности плавного перехода между конструкциями объектно-реляционного отображения и основной системы, более сложное приложение на основе объектно-реляционного отображения может "перейти ниже" на один или несколько уровней для того, чтобы осуществлять взаимодействие с базой данных более специфическим и оптимизированным способом в зависимости от ситуации. По мере развития проекта SQLAlchemy API основной части становился менее пригодным для регулярного использования, в то время, как объектно-реляционное отображение продолжает предоставлять более изощренные и обобщенные шаблоны. Тем не менее, доступность основной части также внесла свою лепту в ранний успех системы SQLAlchemy, так как это обстоятельство позволило пользователям ранних версий достичь гораздо лучших результатов, чем те, которые могли быть достигнуты при использовании разрабатываемой системы объектно-реляционного отображения.

Недостатком подхода, заключающегося в разделении системы на основную часть и объектно-реляционное отображение, является большее количество шагов, требующихся для доставки инструкций. Стандартная реализация интерпретатора языка программирования Python, созданная с использованием языка программирования C, известна значительными затратами ресурсов на осущес-

ствление отдельных вызовов функций, которые являются основной причиной снижения производительности в процессе работы приложения. Традиционные методы обхода этой проблемы заключаются в сокращении цепочек вызовов функций путем перераспределения кода и включения кода в состав существующих функций, а также в замене требующих высокой производительности фрагментов кода на код на языке C. Разработчики SQLAlchemy потратили много лет на использование двух описанных выше методов с целью повышения производительности системы. Однако, продолжающееся распространение интерпретатора PyPy для языка Python может решить оставшиеся проблемы с производительностью без необходимости переписывания большей части внутреннего кода SQLAlchemy на языке C, так как PyPy значительно сокращает потерю производительности при использовании длинных цепочек вызовов функций, применяя inline-функции в ходе процесса JIT-компиляции.

20.3. Использование DBAPI

В основании SQLAlchemy находится подсистема для взаимодействия с базами данных посредством DBAPI. Сам по себе DBAPI представлен не отдельной библиотекой, а исключительно спецификацией. Поэтому реализации спецификации DBAPI доступны для определенных целевых баз данных, таких, как MySQL или PostgreSQL или в качестве альтернативы для определенных адаптеров для баз данных, не совместимых с DBAPI, таких, как ODBC и JDBC.

Использование DBAPI приводит к двум сложностям. Первая сложность заключается в необходимости предоставления простого и полнофункционального фасадного класса дляrudиментарных шаблонов использования DBAPI. Вторая сложность заключается в необходимости обработки значительных отличий специфических реализаций DBAPI, а также используемых систем баз данных.

Система диалектов

Интерфейс, реализуемый в рамках DBAPI является чрезвычайно простым. Его ключевыми компонентами являются сам модуль DBAPI, объект соединения и объект курсора - курсор базы данных представляет контекст определенного запроса и ассоциированные с ним результаты. Простое взаимодействие с этими объектами, направленное на установление соединения с базой данных и извлечение данных из нее, может быть реализовано следующим образом:

```
connection = dbapi.connect(user="user", pw="pw", host="host")
cursor = connection.cursor()
cursor.execute("select * from user_table where name=?", ("jack",))
print "Результирующие столбцы:", [desc[0] for desc in cursor.description]
for row in cursor.fetchall():
    print "Строка:", row
cursor.close()
connection.close()
```

В рамках SQLAlchemy реализован фасадный класс для классического взаимодействия с DBAPI. Точкой входа этого фасадного класса является вызов `create_engine`, с помощью которого устанавливается соединение и собирается конфигурационная информация. В качестве результата выполнения вызова возвращается экземпляр класса `Engine`. Этот объект представляет только способ осуществления запроса через DBAPI, причем последний никогда непосредственно не раскрывается.

Для простого выполнения запросов объект `Engine` предоставляет интерфейс, известный под называнием "интерфейс явного исполнения запросов" ("implicit execution interface"). Работа по созданию и закрытию соединения с базой данных и курсора посредством DBAPI выполняется незаметно для разработчика:

```
engine = create_engine("postgresql://user:pw@host/dbname")
result = engine.execute("select * from table")
print result.fetchall()
```

В версии SQLAlchemy 0.2 был впервые представлен объект `Connection`, позволяющий явно выполнять этапы процесса соединения с базой данных посредством DBAPI:

```
conn = engine.connect()
result = conn.execute("select * from table")
print result.fetchall()
conn.close()
```

Возвращаемый методом `execute` класса `Engine` или `Connection` результат называется `ResultProxy` и предоставляет интерфейс, аналогичный интерфейсу курсора в DBAPI, но с большим набором функций. Объекты `Engine`, `Connection` и `ResultProxy` связаны с модулем DBAPI и являются экземплярами определенного соединения DBAPI и определенного курсора DBAPI соответственно.

На заднем плане объект `Engine` ссылается на объект, называемый `Dialect`. `Dialect` является абстрактным классом, для которого существует множество реализаций, причем каждая из этих реализаций предназначена для работы с определенной комбинацией DBAPI и базы данных. Объект `Connection`, создаваемый на стороне объекта `Engine`, будет ссылаться на этот объект `Dialect` при принятии всех решений, которые могут варьироваться в зависимости от используемых DBAPI и базы данных.

После создания объект `Connection` будет создавать и поддерживать рабочее соединение DBAPI из репозитория, известного, как `Pool`, который также ассоциирован с объектом `Engine`. Репозиторий `Pool` ответственен за создание новых соединений DBAPI и обычно за сохранение их в расположенному в памяти пуле для периодического использования.

В процессе исполнения запроса объектом `Connection` создается дополнительный объект с именем `ExecutionContext`. Этот объект существует с момента исполнения запроса в течение периода существования объекта `ResultProxy`. Он также может быть доступен как специфический подкласс для некоторых комбинаций DBAPI и баз данных.

[Рисунок 20.2](#) иллюстрирует все эти объекты и их взаимоотношения с другими объектами, а также с компонентами DBAPI.

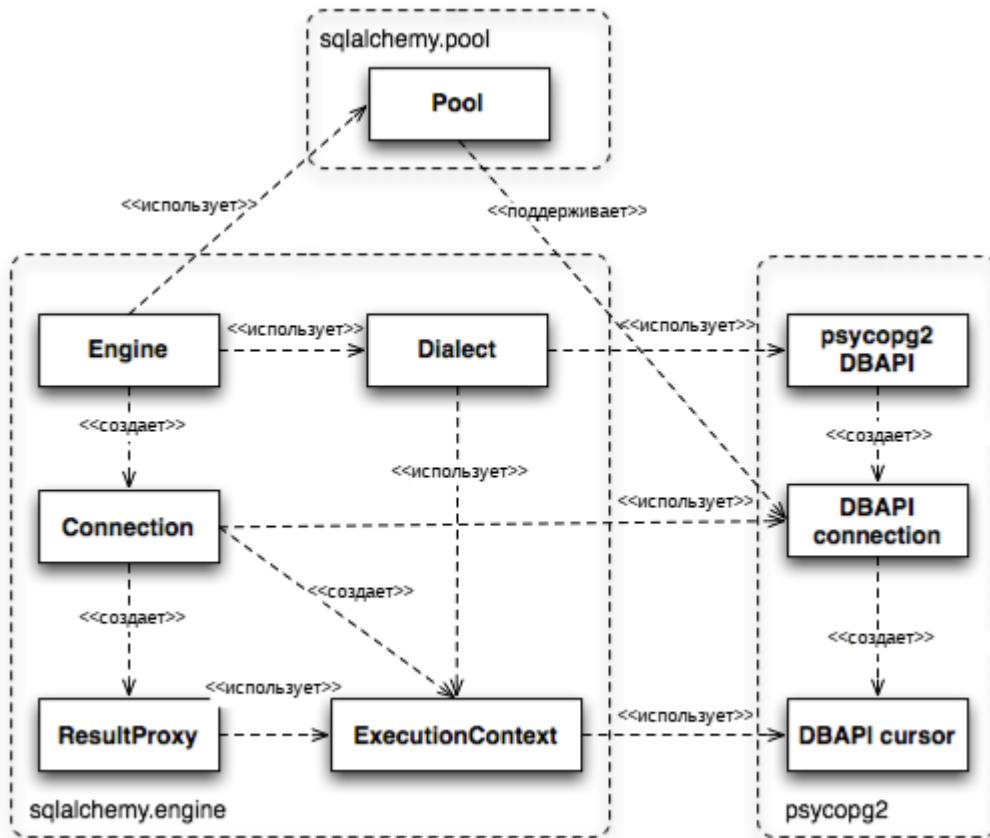


Рисунок 20.2: API объектов Engine, Connection, ResultProxy

Обработка различных интерфейсов DBAPI

Перед рассмотрением задачи обработки различных интерфейсов DBAPI, давайте для начала рассмотрим суть существующей проблемы. Спецификация DBAPI, на данный момент второй версии, написана в форме наборов объявлений API, которые позволяют реализовывать значительно отличающиеся по поведению интерфейсы, а также оставляют большое количество недокументированных областей. В результате существующие реализации DBAPI демонстрируют значительные отличия в некоторых областях, включая возможность или невозможность передачи строк языка Python в кодировке Unicode; способ получения "последнего добавленного идентификатора", являющегося автоматически генерируемым первичным ключом, после выполнения запроса INSERT; а также способ указания и интерпретации граничных значений. Эти интерфейсы также ведут себя индивидуально в зависимости от используемых типов данных, в ситуациях, когда производится обработка бинарных данных, точных числовых данных, дат, логических данных, а также строк в кодировке Unicode.

SQLAlchemy решает эту проблему, допуская различия в классах `Dialect` и `ExecutionContext` путем использования множества уровней подклассов. [Рисунок 20.3](#) иллюстрирует отношение между объектами `Dialect` и `ExecutionContext` в случае использования диалекта psycopg2. Класс `PGDialect` реализует специфичные для базы данных PostgreSQL возможности, такие, как поддержка типа данных ARRAY и каталогов схем; класс `PGDialect_psycopg2` реализует возможности, специфичные для реализации DBAPI psycopg2, включающие обработчики строк в кодировке Unicode и поддержку управления курсором на стороне сервера базы данных.

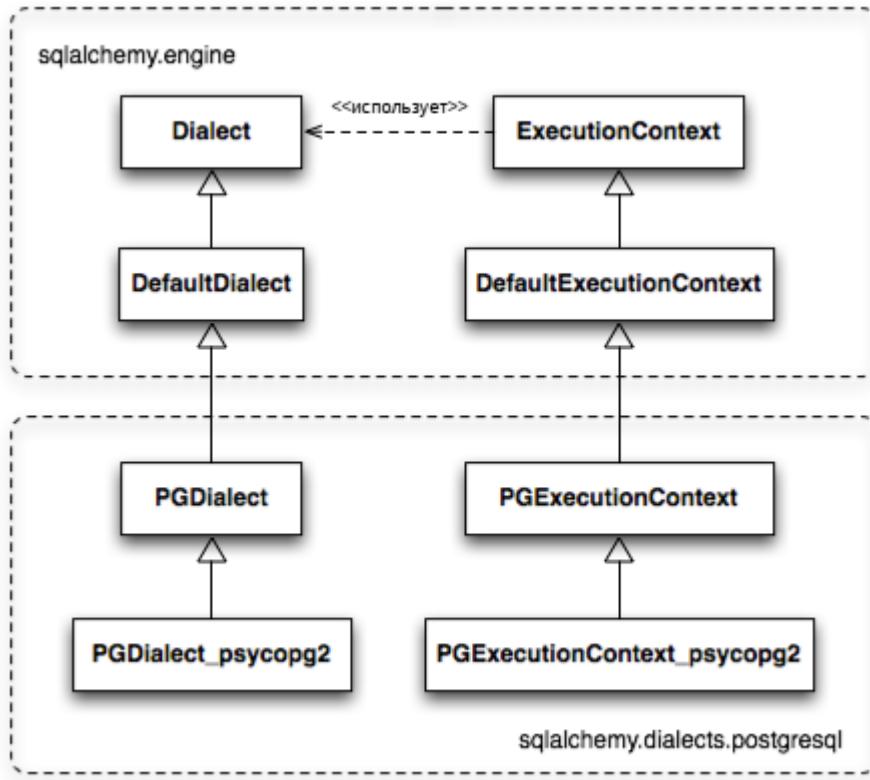


Рисунок 20.3: Простая иерархия классов Dialect/ExecutionContext

Вариант описанного выше шаблона проектирования оправдывает себя при работе с реализацией DBAPI, поддерживающей множество баз данных. Примерами таких реализаций являются `pyodbc`, которая может взаимодействовать с неограниченным количеством баз данных посредством ODBC и `zxjdbcs`, предназначенная только для использования совместно с языком Python и работающая с JDBC. Описанное выше отношение классов реализуется путем использования смешанного класса из пакета `sqlalchemy.connectors`, который реализует особенности работы интерфейса DBAPI, свойственные для множества баз данных. [Рисунок 20.4](#) иллюстрирует стандартные функции класса `sqlalchemy.connectors.pyodbc`, разделяемые между `pyodbc`-специфичными диалектами для баз данных MySQL и Microsoft SQL Server.

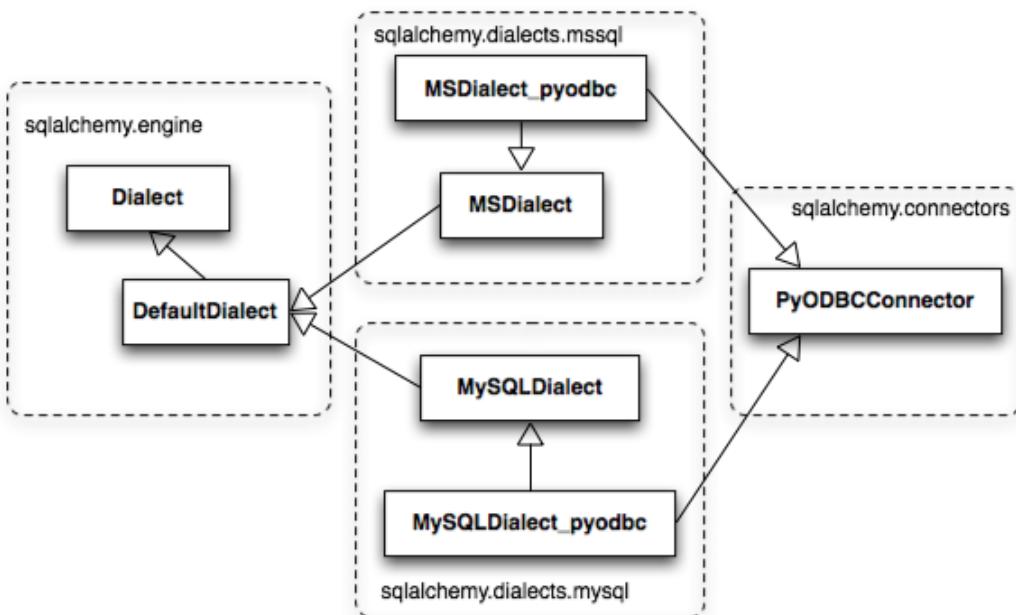


Рисунок 20.4: Стандартные функции DBAPI, разделяемые между иерархиями диалектов

Объекты `Dialect` и `ExecutionContext` предоставляют возможность для описания каждого взаимодействия с базой данных и DBAPI, включая то, как должны форматироваться аргументы функции соединения, а также как должны обрабатываться специальные данные в процессе исполнения запроса. Объект `Dialect` также является фабрикой для компиляции синтаксических конструкций языка SQL, которая осуществляет корректное оформление SQL-запроса для целевой базы данных, а также преобразования типов объектов данных в соответствии с тем, как объекты данных языка Python должны упаковываться и распаковываться при использовании целевых интерфейса DBAPI и базы данных.

20.4. Описание схемы

После установления соединения с базой данных и получения возможности взаимодействия с ней приобретает актуальность задача создания и осуществления манипуляций с зависящими от используемых баз данных SQL-запросами. Для решения этой задачи нам потребуется сформулировать метод создания ссылок на таблицы и столбцы, присутствующие в базе данных - так называемую "схему". Таблицы и столбцы представляют метод организации данных и большинство SQL-запросов состоит из выражений и команд, ссылающихся на эти структуры.

Объектно-реляционное отображение или уровень доступа к данным должен предоставлять программный доступ к возможностям языка SQL; в их основе лежит программная система описания таблиц и столбцов. Это именно то место, где в рамках SQLAlchemy происходит первое жесткое разделение на основную систему и объектно-реляционное отображение путем реализации конструкций `Table` и `Column`, которые описывают структуру базы данных независимо от пользовательского описания класса модели. Обоснованием разделения описания схемы и объектно-реляционного отображения является тот факт, что реляционная схема может быть спроектирована исключительно с использованием терминологии реляционных баз данных, включая платформо-специфичные особенности в случае необходимости без осложнения этого процесса путем введения в него объектно-реляционных концепций - они будут использоваться отдельно. Независимость от компонента объектно-реляционного отображения также подразумевает то, что существующая система описания схем становится настолько же функциональной, как и любая другая объектно-реляционная система, которая могла бы быть реализована на базе основной системы.

Объекты моделей `Table` и `Column` реализуются в области так называемых метаданных (*metadata*), в которой объект коллекции с именем `MetaData` представляет коллекцию объектов `Table`. Структура объектов по большей части реализована в соответствии с описанием "отображения метаданных" ("Metadata Mapping") из книги Martin Flower под названием "*Patterns of Enterprise Application Architecture*". [Рисунок 20.5](#) иллюстрирует некоторые ключевые элементы пакета `sqlalchemy.schema`.

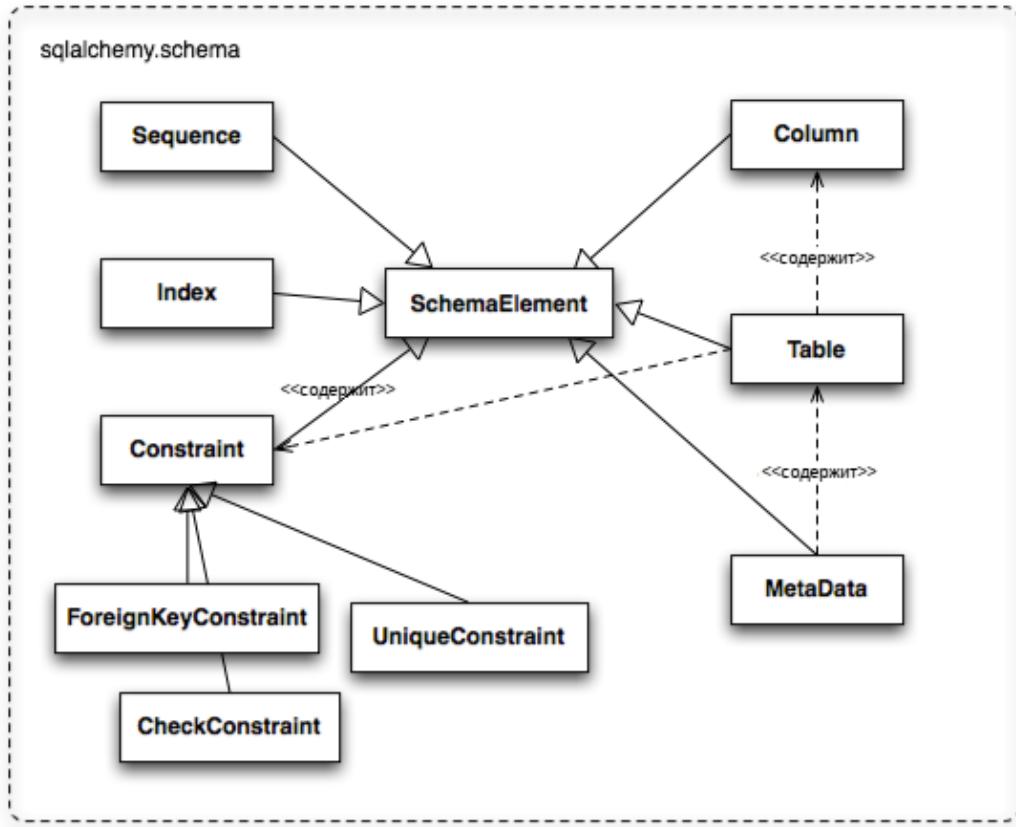


Рисунок 20.5: Базовые объекты пакета sqlalchemy.schema

Объект `Table` представляет имя и другие атрибуты текущей таблицы, присутствующей в целевой схеме. Его коллекция объектов `Column` представляет информацию об именах и типах для определенных столбцов таблицы. Заполненный массив объектов, описывающих ограничения, индексы и последовательности создается для предоставления большего объема информации о таблице, причем некоторые данные непосредственно влияют на принцип работы базы данных и системы формирования SQL-запросов. В частности, объект `ForeignKeyConstraint` является ключевым при определении метода объединения двух таблиц.

Объекты `Table` и `Column` уникальны по сравнению со всеми остальными объектами из пакета для работы со схемами, так как они используют двойное наследование от объектов из пакетов `sqlalchemy.schema` и `sqlalchemy.sql.expression`, работая не только как конструкции уровня обработки схем, но также и как синтаксические единицы языка для создания выражений SQL. Это отношение проиллюстрировано на [Рисунке 20.6](#).

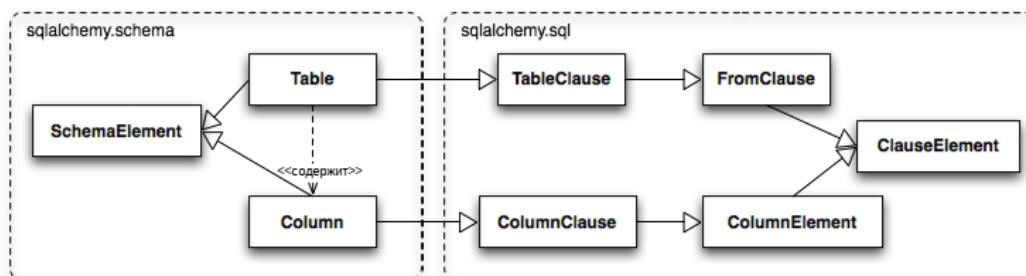


Рисунок 20.6: Двойная жизнь объектов Table и Column

На [Рисунке 20.6](#) мы можем видеть, что объекты `Table` и `Column` наследуются от объектов из мира SQL как специфические формы "вещей из которых вы можете выбрать", известных под именем

`FromClause` и "вещей, которые вы можете использовать в SQL-запросе", известных под именем `ColumnElement`.

20.5. SQL-запросы

В момент начала разработки SQLAlchemy способ генерации SQL-запросов не был ясен. Текстовый язык мог быть хорошим кандидатом; это стандартный подход, лежащий в основе таких широко известных инструментов объектно-реляционного отображения, как HQL из состава Hibernate. В случае использования языка программирования Python, однако, был доступен более занимательный вариант: использование объектов и выражений языка Python для генерации древовидных структур представления запросов, причем возможным было даже изменение назначения операторов языка Python с целью использования их для формирования SQL-запросов.

Хотя рассматриваемый инструмент и не был первым инструментом, выполняющим подобные функции, следует упомянуть о библиотеке SQLBuilder из состава SQLObject от Ian Bicking, которая была использована как образец при создании системы работы с объектами языка Python и операторами, используемыми в рамках языка формирования запросов SQLAlchemy. При использовании данного подхода объекты языка Python представляют лексические части SQL-запроса. Методы этих объектов, также как и перегружаемые операторы, позволяют генерировать новые унаследованные от существующих лексические конструкции. Наиболее часто используемым объектом является представляющий столбец объект `"Column"` - библиотека SQLObject будет представлять такие объекты в рамках класса объектно-реляционного отображения, используя пространство имен с доступом посредством атрибута `.q`; также в SQLAlchemy объявлен атрибут с именем `.c`. Этот атрибут `.c` на сегодняшний день поддерживается и используется для представления элементов основной части, подвергающихся выборке, таких, как объекты, представляющие таблицы и запросы выборки.

Древовидные структуры запросов

Конструкция SQL-запроса в SQLAlchemy очень похожа на структуру, которую вы можете создать в случае разбора готового SQL-запроса - это дерево разбора синтаксических конструкций, отличающееся лишь тем, что вместо формирования на основе строки запроса его создает непосредственно разработчик. Основной тип ветви этого дерева разбора носит имя `ClauseElement`, а [Рисунок 20.7](#) иллюстрирует отношение класса `ClauseElement` и некоторых ключевых классов.

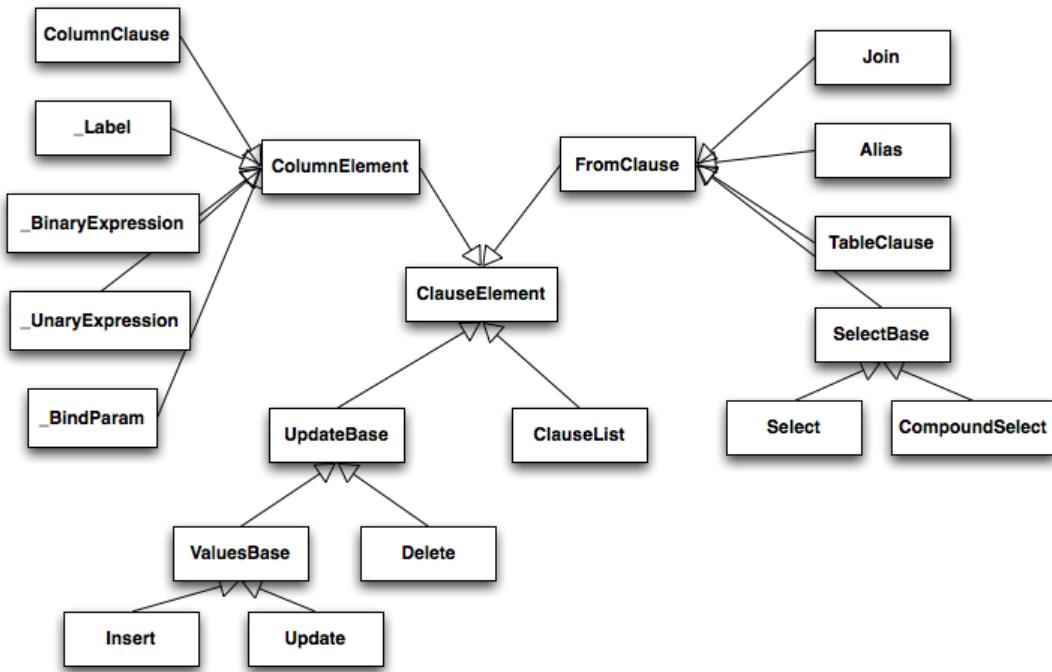


Рисунок 20.7: Базовая иерархия классов запроса

Путем использования функций-конструкторов, методов и перегруженных функций операторов языка Python, структура аналогичного следующему запросу:

```
SELECT id FROM user WHERE name = ?
```

может быть следующим образом сформирована с помощью средств языка Python:

```
from sqlalchemy.sql import table, column, select
user = table('user', column('id'), column('name'))
stmt = select([user.c.id]).where(user.c.name=='ed')
```

Структура описанной выше конструкции `select` показана на [Рисунке 20.8](#). Следует отметить, что представление строкового значения 'ed' находится внутри конструкции `_BindParam`, что приводит к трактовке его как маркера параметра ограничения, для обозначения которого в строке SQL-запроса используется знак вопроса.

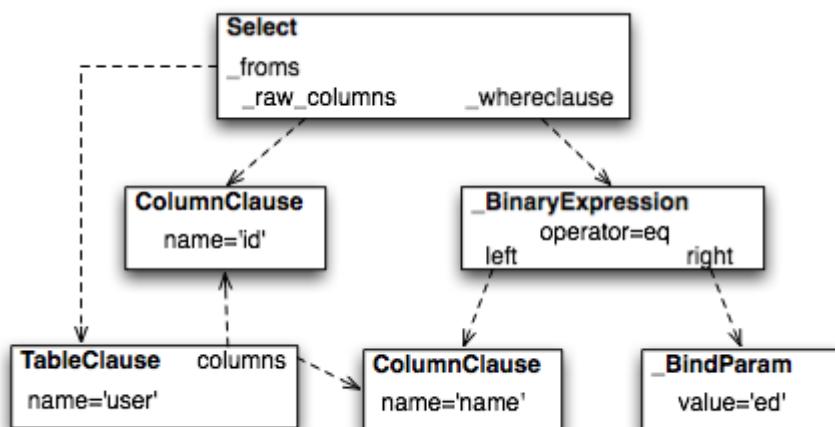


Рисунок 20.8: Пример древовидного представления запроса

При рассмотрении древовидной диаграммы можно увидеть, что в ходе простого обратного обхода узлов может быть быстро получен сформированный SQL-запрос, при этом мы сможем ознакомиться с упомянутым процессом гораздо подробнее в разделе, посвященном компиляции запросов.

Подход к использованию операторов языка Python

В SQLAlchemy подобный следующему запрос:

```
column('a') == 2
```

возвращает не логический результат `True` или `False`, а конструкцию SQL-запроса. Это делается для того, чтобы выполнить перегрузку операторов с помощью специальных функций для управления операторами языка Python, т.е., таких методов, как `__eq__`, `__ne__`, `__le__`, `__lt__`, `__add__`, `__mul__`. Узлы структуры, связанные со столбцами, предоставляют возможность работы с перегруженными операторами языка Python путем использования смешанного класса с именем `ColumnOperators`. После использования возможности перегрузки операторов запрос `column('a') == 2` эквивалентен следующему коду:

```
from sqlalchemy.sql.expression import _BinaryExpression
from sqlalchemy.sql import column, bindparam
from sqlalchemy.operators import eq

(BinaryExpression(
    left=column('a'),
    right=bindparam('a', value=2, unique=True),
    operator=eq
))
```

Конструкция `eq` на самом деле является функцией из встроенного модуля `operator`. Представление операторов в виде объектов (т.е., `operator.eq`) вместо строк (т.е., `=`) позволяет задавать строковое представление запроса во время компиляции, когда имеется информация о диалекте используемой базы данных.

Компиляция

Главным классом, ответственным за преобразование древовидных представлений SQL-запросов в текстовый формат SQL-запросов является класс с именем `Compiled`. Этот класс имеет два основных подкласса, `SQLCompiler` и `DOLCompiler`. Класс `SQLCompiler` выполняет операции преобразования запросов `SELECT`, `INSERT`, `UPDATE` и `DELETE`, обобщенно классифицируемых как элементы языка запросов данных (`data query language - DQL`) и языка для манипуляций с данными (`data manipulation language - DML`), в то время, как класс `DDLCompiler` выполняет операции преобразования различных запросов `CREATE` и `DROP`, классифицируемых как элементы языка описания данных (`data definition language - DDL`). Существует дополнительная иерархия классов, предназначенная для реализации функций преобразования представлений строк на основе типов, задаваемых в рамках класса `TypeCompiler`. Отдельные диалекты предоставляют свои собственные подклассы всех трех типов классов компилятора с целью поддержки специфических для используемой базы данных аспектов языка SQL-запросов. На [Рисунке 20.9](#) представлен обзор этой иерархии классов, сформированной для работы с диалектом PostgreSQL.

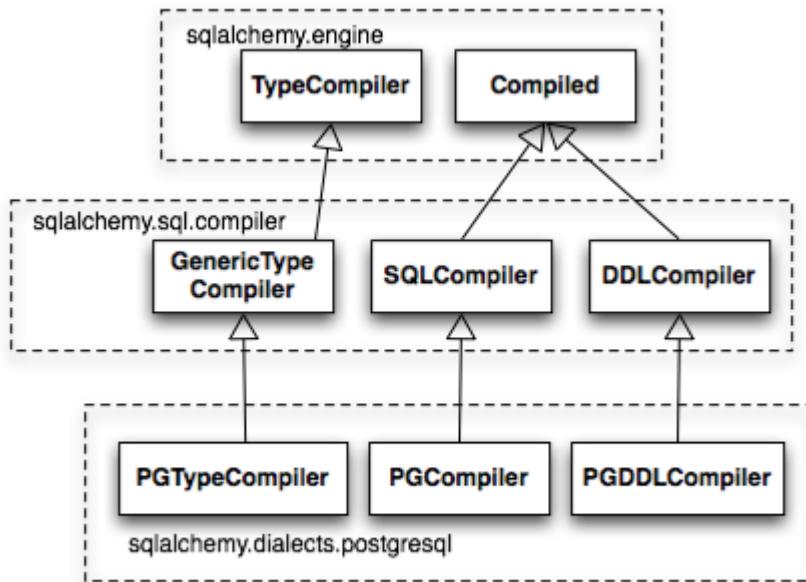


Рисунок 20.9: Иерархия классов компилятора, включающая специфичные для PostgreSQL реализации классов

Подклассы класса `Compiled` описывают наборы `visit`-методов, на каждый из которых ссылается определенный подкласс класса `ClauseEvent`. Производится обход иерархии классов узлов `ClauseEvent`, после чего запрос формируется путем рекурсивного объединения строковых результатов, возвращаемых каждой из `visit`-функций. По мере выполнения этой работы объект `Compiled` изменяет состояние в зависимости от имен анонимных идентификаторов, имен граничных параметров, а также находящихся среди прочих параметров в составе запроса подзапросов, каждый из которых влияет как на результат генерации строки SQL-запроса, так и на конечный набор граничных параметров с их значениями по умолчанию. [Рисунок 20.10](#) иллюстрирует процесс использования `visit`-методов для получения текстовых фрагментов запроса.

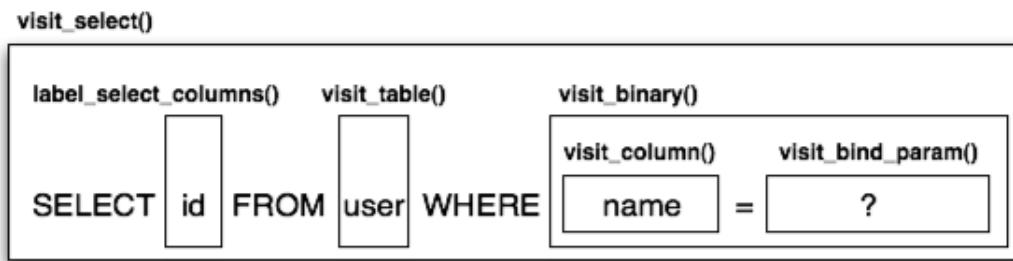


Рисунок 20.10: Иерархия вызовов функций в ходе компиляции запроса

Заполненная данными структура `Compiled` содержит завершенную строку SQL-запроса и набор граничных значений. Эти данные преобразуются с помощью класса `ExecutionContext` в формат, ожидаемый методом `execute` реализации интерфейса DBAPI, который использует предположения об обязательном использовании кодировки Unicode в рамках объекта запроса, о типе коллекций, используемой для хранения граничных значений, а также о специфике того, как граничные условия должны преобразовываться в представления, подходящие для использования совместно с реализацией интерфейса DBAPI и используемой базой данных.

20.6. Отображение классов при использовании объектно-реляционного отображения

Переключим наше внимание на объектно-реляционное отображение. Первой целью является использование описанной нами системы таблиц метаданных для предоставления возможности переноса функций заданного пользователем класса на коллекцию столбцов в таблице базы данных. Второй целью является предоставление возможности описания отношений между заданными пользователем классами, которые будут основываться на отношениях между таблицами в базе данных.

В SQLAlchemy такая связь называется "отображением", что соответствует широко известному шаблону проектирования с названием "DataMapper", описанному в книге Martin Flower с названием "*Patterns of Enterprise Architecture*". В целом, система объектно-реляционного отображения SQLAlchemy была разработана с применением большого количества приемов, которые описал в своей книге Martin Flower. Она также подверглась значительному влиянию со стороны известной системы реляционного отображения Hibernate для языка программирования Java и продукта SQLObject для языка программирования Python от Ian Bicking.

Классическое отображение против декларативного отображения

Мы используем термин "классическое отображение" для указания на систему объектно-реляционного отображения данных для существующего пользовательского класса в рамках SQLAlchemy. Эта форма отображения использует объект `Table` и заданный пользователем класс для формирования двух связанных с помощью функции с именем `mapper`, но при этом отдельных, примитивов. Как только функция `mapper` применяется по отношению к заданному пользователем классу, класс приобретает новые атрибуты, соответствующие столбцам таблицы:

```
class User(object):
    pass

mapper(User, user_table)

# теперь у объекта User есть атрибут ".id"
User.id
```

Функция `mapper` также может добавлять другие атрибуты классу, включая атрибуты, соответствующие как ссылкам на другие типы объектов, так и на произвольные SQL-запросы. Процесс добавления произвольных атрибутов классу в мире Python известен под названием "monkeypatching"; однако, так как мы выполняем его на основе данных и не привольным образом, суть процесса гораздо лучше может быть обозначена термином "доработка класса" ("*class instrumentation*").

Современный подход к работе с SQLAlchemy базируется на использовании декларативного расширения, которое представляет собой систему конфигурации, имеющую сходство с известной, похожей на `active record` системой декларативного описания классов, используемой во множестве других инструментов объектно-реляционного отображения. В этой системе конечный пользователь явно задает описание атрибута в процессе создания описания класса, каждое из которых представляет атрибут класса, который должен быть использован при создании отображения. Объект `Table` в большинстве случаев не упоминается ни явно, ни при использовании функции `mapper`; явно упоминается только пользовательский класс, объекты `Column` и другие относящиеся к отображению атрибуты:

```
class User(Base):
    __tablename__ = 'user'
    id = Column(Integer, primary_key=True)
```

При рассмотрении приведенного выше фрагмента кода может показаться, что добавление атрибутов в класс осуществляется непосредственно в строке `id = Column()`, но на самом деле это не так.

Декларативное расширение использует метакласс Python, с помощью которого очень удобно выполнять серии операций каждый раз, когда новый класс впервые декларируется с целью генерации нового объекта `Table` на основе декларации и передачи его функции `mapper` вместе с классом. После этого функция `mapper` выполняет работу точно таким же образом, добавляя набор атрибутов в класс, причем в данном случае используется атрибут `id`, а также заменяя атрибуты, которые были добавлены ранее. Ко времени окончания процесса инициализации метакласса (т.е., к моменту, когда поток выполнения покидает блок описания класса `User`), объект `Column` с атрибутом `id` перемещается в новый объект `Table` и `User.id` заменяется на новый атрибут, специфичный для отображения.

Всегда считалось, что система SQLAlchemy должна иметь четкую декларативную форму конфигурации. Однако, создание декларативного расширения задерживалось из-за продолжающейся работы по стабилизации механизмов классического отображения. Ранее существовало временное расширение под названием `ActiveMapper`, которое впоследствии было преобразовано в проект Elixir. Оно позволяло повторно объявлять конструкции отражения в рамках декларативной системы более высокого уровня. Задача декларативного расширения была противоположной задаче проекта Elixir и заключалась в создании мощной абстракции путем создания системы, которая практически полностью сохраняет классические концепции SQLAlchemy в области отображения данных, допуская реорганизацию метода использования для сокращения объема отладочной информации и повышения совместимости с расширениями уровня классов по сравнению с возможностями классических систем отображения.

Вне зависимости от того, используется ли классическое или декларативное отображение, используемый для отображения класс получает новые возможности, которые позволяют ему работать с синтаксическими конструкциями SQL, представленными в форме атрибутов. Изначально принцип использования специального атрибута в качестве источника SQL-запросов для столбцов в SQLAlchemy соответствовал принципу использования такового в `SQLObject` и заключался в использовании при работе с SQLAlchemy атрибута `.c`, как показано в этом примере:

```
result = session.query(User).filter(User.c.username == 'ed').all()
```

Однако, в версии 0.4 SQLAlchemy эти функции были возложены непосредственно на отображаемые атрибуты:

```
result = session.query(User).filter(User.username == 'ed').all()
```

Это изменение метода доступа к атрибутам оказалось значительным усовершенствованием, так как оно позволило аналогичным объектам столбцов объектам находиться в классе для достижения дополнительных, специфичных для класса возможностей, которые не доступны для классов, наследуемых напрямую от находящегося уровнем ниже класса `Table`. Оно также позволило интегрировать различные типы атрибутов классов, такие, как атрибуты, ссылающиеся непосредственно на столбцы таблицы, атрибуты, ссылающиеся на SQL-запросы, созданные на основе этих столбцов, а также атрибуты, ссылающиеся на соответствующий класс. Наконец, оно позволило достичь соответствия между классом для отображения и экземпляром этого класса для отображения, в котором тот же атрибут может быть предназначен для работы с другой информацией в зависимости от родительского класса. Атрибуты классов возвращают SQL-запросы, в то время, как атрибуты экземпляров классов возвращают реальные данные.

Анатомия отображения

Атрибут `id`, который был привязан к нашему классу `User`, является объектом, тип которого известен в рамках языка программирования Python как дескриптор (*descriptor*), причем этот объект поддерживает методы `__get__`, `__set__` и `__del__`, которые интерпретатор Python позволяет ис-

пользовать во всех операциях, связанных с этим классом или его экземпляром. Реализация в рамках SQLAlchemy известна под именем `InstrumentedAttribute` и мы продемонстрируем механизмы, скрытые за этим фасадным классом, в ходе рассмотрения следующего примера. Начиная работу с описания класса `Table`, а также пользовательского класса, мы начинаем формирование отображения только для одного столбца, а также используем функцию `relationship`, которая задает ссылку на связанный класс:

```
user_table = Table("user", metadata,
    Column('id', Integer, primary_key=True),
)

class User(object):
    pass

mapper(User, user_table, properties={
    'related':relationship(Address)
})
```

После создания отображения структура относящихся к классу объектов будет соответствовать представленной на [Рисунке 20.11](#).

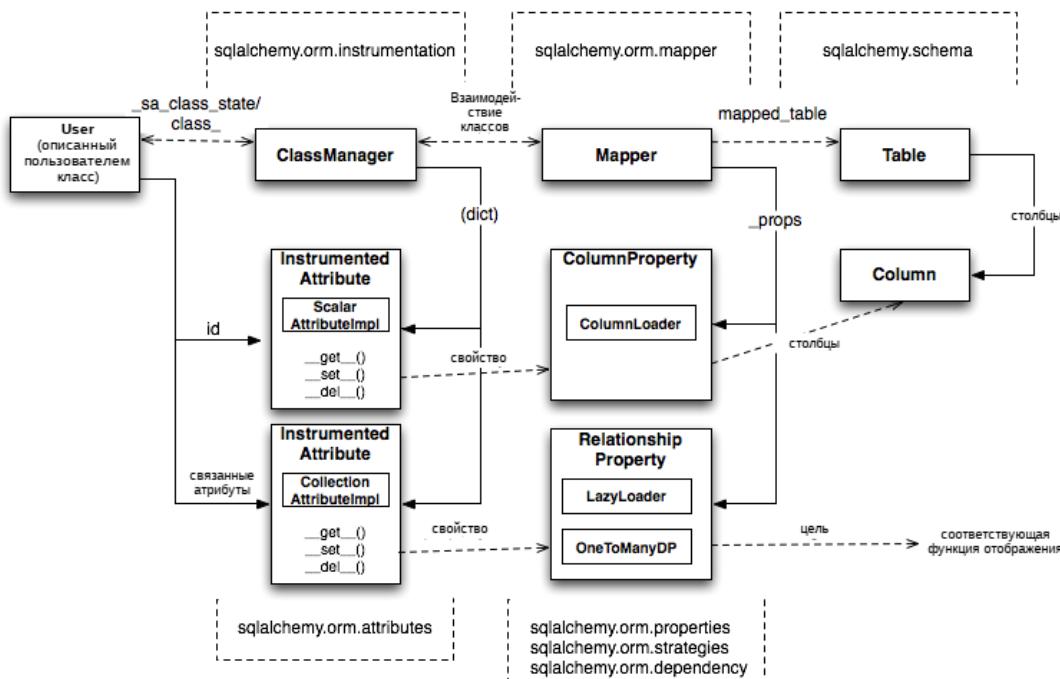


Рисунок 20.11: Анатомия отображения

На рисунке проиллюстрировано созданное с помощью SQLAlchemy отображение, использующее два отдельных уровня для осуществления взаимодействия заданного пользователем класса и метаданных таблицы, с которой он связан. Инструментарий, предназначенный для реализации функций класса представлен в левой части рисунка, в то время, как функции для работы с базой данных и SQL-запросами - в правой части. Основной используемый шаблон проектирования подразумевает то, что композиция объектов используется для разделения моделей поведения объектов, а наследование объектов используется для разделения вариаций поведения объектов в рамках определенной роли.

В области инструментария класса следует выделить класс `ClassManager`, который связан с используемым для отображения классом, причем каждый объект `InstrumentedAttribute` из его коллекции связан с каждым отображаемым атрибутом класса. `InstrumentedAttribute` также является упомянутым ранее общедоступным дескриптором языка Python и позволяет формировать SQL-запросы при использовании совместно с запросами на основе классов (т.е., `User.id==5`). При

разговоре об экземпляре класса `User`, следует упомянуть о том, что объект `InstrumentedAttribute` делегирует поддержку атрибута объекту `AttributeImpl`, который является одним из нескольких аналогичных объектов, выбранным в соответствии с типом представляемых данных.

Со стороны отображения объект `Mapper` представляет связь заданного пользователем класса и выбираемого элемента базы данных, чаще всего объекта таблицы `Table`. Объект `Mapper` поддерживает коллекцию объектов, известных как `MapperProperty` и соответствующих каждому из атрибутов, которые отвечают за представление определенного атрибута в SQL-запросе. Наиболее часто встречающимися вариантами объектов `MapperProperty` являются объекты `ColumnProperty`, представляющие столбцы в SQL-запросе, а также объекты `RelationshipProperty`, представляющие связь с другим классом отображения.

Объект `MapperProperty` делегирует функции загрузки атрибутов, включая функции преобразования атрибутов в фрагменты SQL-запроса и функции их извлечения из строк результатов запросов, объекту `LoaderStrategy`, который также может быть представлен несколькими вариантами. Различные объекты `LoaderStrategies` устанавливают методы загрузки атрибута в соответствии с вариантами: *deferred* (отложенная загрузка), *eager* (быстрая загрузка) или *immediate* (немедленная загрузка). Стандартная версия поведения выбирается во время конфигурации отображения, при этом дополнительным вариантом является использование альтернативной стратегии в момент выполнения запроса. Объект `RelationshipProperty` также ссылается на объект `DependencyProcessor`, который устанавливает метод обработки зависимостей между отображениями и метод синхронизации атрибутов для применения во время сохранения данных. Выбор объекта `DependencyProcessor` основывается на параметрах отношения родительских и целевых элементов, связанных друг с другом.

Структура `Mapper/RelationshipProperty` формирует граф, в котором объекты `Mapper` являются узлами, а объекты `RelationshipProperty` - ориентированными ребрами. После того, как полный набор функций отображения декларируется приложением, наступает этап отложенной "инициализации", известный как процесс конфигурации (*configuration*). Данные, собираемые в ходе выполнения этого процесса, главным образом используются каждым объектом `RelationshipProperty` для уточнения параметров, используемых функциями отображения родительских (*parent*) и целевых (*target*) объектов, причем также производится выбор объекта `AttributeImpl` наряду с объектом `DependencyProcessor`. Этот граф является ключевой структурой данных, используемой при выполнении операции объектно-реляционного отображения. Он участвует в операциях, использующих так называемые "каскады", которые устанавливают последовательность выполнения функций на основе последовательностей объектов в операциях запросов, в которых связанные объекты и коллекции объектов "быстро" одновременно загружаются, а также в процессах сохранения данных объектов, когда график зависимостей для всех объектов создается до того, как будут завершены этапы освобождения выделенных для хранения данных ресурсов.

20.7. Методы выполнения запросов и загрузки данных

`SQLAlchemy` инициирует все связанные с загрузкой объектов действия с помощью объекта с именем `Query`. Стандартный процесс инициализации объекта `Query` начинается с включения примитивов (*entities*), которые формируют список классов, используемых для отображения и/или индивидуальных SQL-запросов, которые должны быть выполнены. Он также содержит ссылку на объект `Session`, который представляет соединение с одной или большим количеством баз данных, а также кэш данных, который был собран при осуществлении транзакций в ходе использования этих соединений. Ниже представлен элементарный пример использования этих объектов:

```
from sqlalchemy.orm import Session
session = Session(engine)
```

```
query = session.query(User)
```

Мы создаем объект `Query`, который использует экземпляры класса `User`, относящиеся к новому объекту сессии `Session`, который мы создали. Объект `Query` использует генеративный шаблон проектирования `builder` таким же образом, как и описанная ранее конструкция `select`, где дополнительные критерии и модификаторы ассоциировались с конструкциями выражений для выполнения одного вызова метода в каждый момент времени. Когда итеративная операция выполняется в отношении объекта `Query`, он создает конструкцию SQL-запроса `SELECT`, выполняет этот запрос с помощью базы данных, после чего интерпретирует результирующий набор строк как результатирующие данные объектно-реляционного отображения в соответствии с набором изначально запрашиваемых элементов.

Объект `Query` производит жесткое разделение между частями операции, относящимися к формированию SQL-запросов (*SQL rendering*) и частями операции, относящимися к загрузке данных (*data loading*). Первый случай относится к созданию запроса `SELECT`, а второй - к интерпретации полученных после выполнения SQL-запроса строк как конструкций объектно-реляционного отображения. Фактически данные могут обрабатываться без выполнения шага, заключающегося в формировании SQL-запроса, так как объект `Query` может интерпретировать результаты выполнения текстового запроса, сформированного пользователем вручную.

И в процессе формирования SQL-запроса, и в процессе загрузки данных используется рекурсивный обход графа, образованного наборами объектов `Mapper`, причем каждый содержащий данные столбца или SQL-запроса объект `ColumnProperty` рассматривается как узел, а каждый объект `RelationshipProperty`, который включается в запрос с помощью так называемой стратегии "eager-load" - как ребро графа, ведущее к другой представленной объектом `Mapper` вершине. Обход и выполнение действия при достижении каждого из узлов в конечном счете является задачей каждого объекта `LoaderStrategy`, ассоциированного с каждым объектом `MapperProperty`, осуществляющим добавление столбцов и объединений к запросу `SELECT`, создаваемому на этапе формирования запроса и создающему функции языка Python для обработки результирующих строк на этапе загрузки данных.

Все функции языка Python, создающиеся на этапе загрузки данных, получают по строке из базы данных по мере извлечения и в результате производят возможные изменения состояния отображеного атрибута в памяти. Они создаются для определенного атрибута в соответствии с условием, базирующимся на исследовании первой строки из полученного набора результирующих данных, а так же на параметрах загрузки. Если загрузка атрибута не производится, пригодной для вызова функции не создается.

[Рисунок 20.12](#) иллюстрирует обход нескольких объектов `LoaderStrategy` при сценарии загрузки данных в соответствии со стратегией "joined eager loading" с указанием на их объединение с формируемым SQL-запросом, который появляется в ходе вызова метода `_compile_context` объекта `Query`. На нем также показан процесс генерации функций обработки строк (*row population functions*), которые получают результирующие строки и задают отдельные атрибуты объектов, причем сам этот процесс инициируется с помощью метода `instances` объекта `Query`.

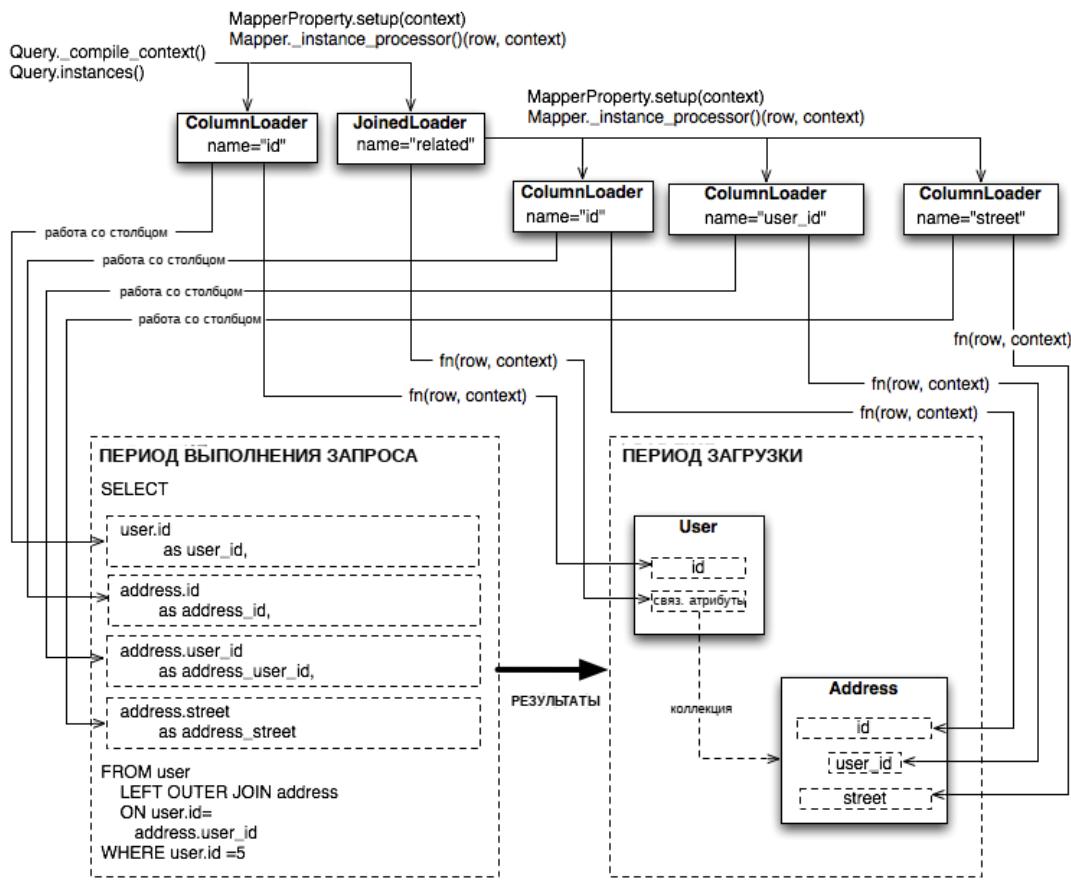


Рисунок 20.12: Обход объектов при использовании стратегий загрузки, включая стратегию "joined eager load"

Ранее в SQLAlchemy использовался подход, заключающийся в сборе результатов, полученных после вызовов фиксированных, ассоциированных с каждой стратегией методов объектов для получения каждой строки таблицы и выполнения соответствующих действий. Система загрузки с возможностью вызова функций была впервые представлена в версии 0.5 и позволила значительно повысить производительность системы, так как большое количество решений в отношении обработки строк таблицы могло быть принято прямо перед началом их обработки вместо повторного принятия решений для каждой строки, а также большое количество не имеющих эффекта вызовов функций могло быть устранено.

20.8. Сессия и индивидуальное отображение

В SQLAlchemy объект `Session` представляет публичный интерфейс для непосредственного использования объектно-реляционного отображения, позволяющий загружать и хранить на постоянной основе данные. Он является отправной точкой для выполнения запросов и операций сохранения данных для заданного соединения с базой данных.

В дополнение к поддержке соединения с базой данных, объект `Session` поддерживает активный список ссылок для набора, состоящего из всех отображенных элементов, которые присутствуют в памяти и относятся к сессии, представленной данным объектом `Session`. Таким образом, класс `Session` является фасадным классом, использующим шаблоны проектирования индивидуального отображения (*identity map*) и рабочей единицы (*unit of work*), которые также описаны Martin Flower. Индивидуальное отображение позволяет поддерживать уникальное в рамках базы данных отображение всех объектов определенной представленной объектом `Session` сессии, исключая проблемы, связанные с наличием дубликатов элементов. Рабочая единица является надстройкой над индивидуальным отображением, реализующей систему автоматизации процесса сохранения на постоянной основе всех изменений состояния базы данных наиболее эффективным способом из

возможных. Сам этап сохранения данных известен как "этап записи данных", причем в современных версиях SQLAlchemy он обычно выполняется автоматически.

История разработки

Класс `Session` начал свое существование как наиболее закрытая система, ответственная за выполнение единственной задачи, заключающейся в сохранении данных. Процесс сохранения данных подразумевает выполнение SQL-запросов с помощью базы данных в соответствии с изменениями состояния объектов, отслеживаемых системой рабочей единицы и, в связи с этим, синхронизации текущего состояния базы данных с содержимым памяти. Сохранение данных всегда было одной из наиболее сложных операций, выполняемых SQLAlchemy.

Выполнение операции сохранения данных (*flush*) в ранних версиях начинается после вызова метода с именем `commit`, который присутствовал у явно созданного в рамках потока объекта с именем `objectstore`. Во время использования нами версии 0.1 SQLAlchemy не было особой надобности в вызове метода `Session.add`, как, впрочем, и не было какой-либо четкой концепции класса сессии `Session` вообще. Единственными доступными пользователю операциями были операции создания функций для формирования отображений, операции создания новых объектов, операции модификации существующих объектов, загруженных в ходе выполнения запросов (в этом случае сами запросы выполнялись непосредственно через объект `Mapper`), после чего все данные должны были сохраняться с помощью команды `objectstore.commit`. Пул объектов для набора операций был безусловно глобальным в рамках модуля и безусловно локальным в рамках потока.

Модель `objectstore.commit` была популярна среди группы пользователей ранних версий, но отсутствие гибкости этой модели быстро привело к потере актуальности. Пользователи, начавшие работать с новыми версиями SQLAlchemy, иногда жаловались на необходимость создания фабрики и, возможно, реестра для объектов `Session`, а также на необходимость сохранения своих объектов, организованных в рамках одного объекта `Session` в каждый момент времени, но этот подход гораздо предпочтительнее существовавшего ранее, при котором принцип работы всех элементов системы был четко задан. Присущие использованному в версии 0.1 шаблону проектирования пользовательские качества все еще в большей степени присутствуют в современных версиях SQLAlchemy, в которых реализован реестр сессий, обычно настроенный для использования локального пространства потока.

Сам объект `Session` был представлен только в версии 0.2 SQLAlchemy и был смоделирован в общих чертах по аналогии с объектом `Session` из состава Hibernate. Эта версия содержала интегрированный механизм контроля транзакций, в рамках которого объект `Session` мог помещаться в транзакцию с помощью метода `begin`, а завершение транзакции осуществлялось с помощью метода `commit`. Метод `objectstore.commit` был переименован в `objectstore.flush` и новые объекты `Session` могли создаваться в любой момент. Сам объект `Session` был отделен от другого объекта с именем `UnitOfWork`, который остался приватным объектом, ответственным за выполнение операции сохранения данных.

Хотя процесс сохранения данных и был изначально реализован в рамках явно вызываемого пользователем метода, в версии 0.4 SQLAlchemy была представлена концепция автоматического сохранения изменений (*autoflush*), которая предполагала, что сохранение данных будет осуществляться непосредственно перед каждым запросом. Преимущество автоматического сохранения данных заключается в том, что SQL-запрос, выполняемый в результате запроса всегда имеет доступ со стороны реляционной базы данных к данным, присутствующим в памяти, так как все изменения были переданы. Ранние версии SQLAlchemy не могли включать эту возможность, так как стандартным шаблоном проектирования оговаривалось, что сохранение данных должно сопровождаться полным сохранением изменений. Но в момент представления концепции автоматического сохранения данных была реализована сопутствующая ей возможность под названием "транзакционная сессия" ("*transactional session*") в рамках объекта `Session`, заключающаяся в предоставле-

нии объекта `Session`, который должен был автоматически начинать транзакцию, продолжавшуюся до того момента, когда пользователь явно вызывал метод `commit`. После реализации этой возможности метод `flush` больше не записывал данные для сохранения и мог вызываться автоматически. Сейчас объект `Session` позволяет осуществлять пошаговую синхронизацию между данными состояния в памяти и данными состояния SQL-запроса путем сохранения данных при необходимости без постоянного сохранения данных до момента явного вызова метода `commit`. Такое поведение фактически точно повторяет поведение системы Hibernate для языка программирования Java. Однако, этот стиль работы был реализован в SQLAlchemy благодаря использованию в качестве примера системы Storm ORM для языка программирования Python, представленной в момент существования версии 0.3 системы SQLAlchemy.

В версии 0.5 была улучшена работа с транзакциями и представлена схема истечения срока действия транзакции (*post-transaction expiration*); после каждого использования методов `commit` и `rollback` по умолчанию истекал срок действия всех данных состояния в рамках объекта `Session` (они удалялись) и они должны были снова извлекаться в ходе выполнения последующих SQL-запросов или тогда, когда доступ к атрибутам оставшегося набора объектов с истекшим сроком действия осуществляется из контекста новой транзакции. Изначально система SQLAlchemy была спроектирована в соответствии с предположением о том, что запросы SELECT должны безусловно выполняться так мало раз, как это возможно. Стратегия истечения срока действия данных при их записи внедрялась медленно именно по этой причине; однако, она полностью решала проблему хранения в рамках объекта `Session` устаревшей копии полученных после транзакции данных, для обновления которой не было предложено простого, не требующего полномасштабного повторного создания набора уже загруженных объектов способа. Сначала казалось, что эта проблема не имеет разумного решения, так как момент, после которого объект `Session` должен считать текущие данные состояния устаревшими и, следовательно, использовать набор ресурсоемких запросов SELECT при следующей попытке доступа к данным, не был очевиден. Однако, как только объект `Session` начал постоянно использоваться в рамках транзакций, момент завершения транзакции стал естественным четким моментом истечения срока действия данных, так как природа транзакции с высокой степенью изоляции состоит в том, что она не может получить доступ к новым данным до того, как ее данные будут записаны, либо она будет отменена. Различные базы данных и конфигурации, конечно же, характеризуются различными степенями изоляции транзакций, включая отсутствие транзакций как таковых. Эти режимы работы полностью допустимы при использовании модели истечения срока действия данных SQLAlchemy; разработчик должен заботиться только о том, чтобы низкая степень изоляции не привела к раскрытию неизолированных изменений в рамках сессии в том случае, когда множество сессий использует одни и те же строки. Эта ситуация ни коим образом не отличается от ситуации, которая может произойти при непосредственном использовании двух соединений с базой данных.

Обзор сессии

[Рисунок 20.13](#) иллюстрирует объект `Session` и основные структуры, взаимодействующие с ним.

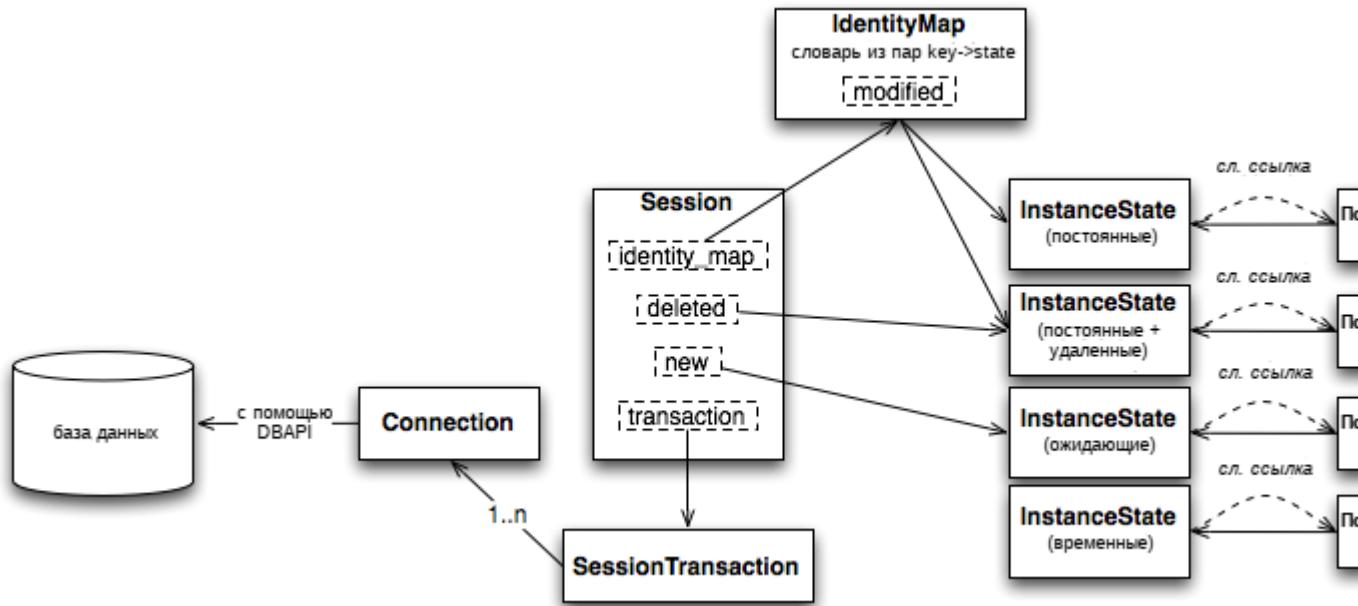


Рисунок 20.13: Обзор объекта Session

Общедоступными объектами на рисунке выше является сам объект `Session`, а также коллекция пользовательских объектов, каждый из которых является экземпляром класса, используемого для создания отображения. Здесь мы можем увидеть, что используемые для отображения объекты ссылаются на конструкцию из состава SQLAlchemy с именем `InstanceState`, которая отслеживает состояние отдельного объектно-реляционного отображения, включая ожидающие операции изменения атрибутов, а также факт истечения срока действия атрибутов. Объект `InstanceState` является инструментарием для работы с атрибутами на уровне экземпляра класса, описанным в предыдущем разделе под названием *"Анатомия отображения"* и соответствующим объекту `ClassManager` на уровне класса, который позволяет поддерживать состояние словаря используемого для создания отображения объекта (т.е., атрибута `__dict__`, описанного в рамках языка программирования Python) на стороне ассоциированных с классом объектов `AttributeImpl`.

Отслеживание состояния

Объект `IdentityMap` позволяет создавать отображение индивидуальных данных базы данных для объектов `InstanceState`, которые в свою очередь используются теми объектами, которым требуются эти индивидуальные данные, называемые также постоянными (*persistent*). Стандартная реализация объекта `IdentityMap` взаимодействует с объектом `InstanceState` для самостоятельного управления объемом занятой памяти путем удаления созданных пользователем отображений в тех случаях, когда удаляются все жесткие ссылки на эти отображения - таким образом, этот объект функционирует аналогично объекту `weakValueDictionary` из состава Python. Объект `Session` защищает набор всех объектов с пометкой "устаревший" ("dirty") или "удаленный" ("deleted"), а также охраняет объекты с пометкой "новый" ("new") от механизма сборки мусора путем создания жестких ссылок на эти объекты в случае ожидания их изменений. Все жесткие ссылки удаляются после выполнения операции сохранения данных.

Объект `InstanceState` также выполняет критичную задачу, заключающуюся в поддержании "списка изменений" для атрибутов определенного объекта с использованием системы перемещения данных при изменении, которая сохраняет "данные предыдущего состояния" определенного атрибута в словаре с именем `committed_state` перед использованием переданного значения для изменения значения в словаре атрибутов объекта. Во время выполнения операции сохранения изменений содержимое словаря `committed_state`, а также ассоциированного с объектом словаря `__dict__` сравниваются с целью формирования набора измененных данных для каждого из объектов.

В случае коллекций отдельный пакет с именем `collections` осуществляет координацию работы системы объектов `InstrumentedAttribute/InstanceState` для поддержания функционирования коллекции изменений для определенной коллекции объектов, используемых для отображения. Такие стандартные классы языка Python, как `set`, `list` и `dict` перед использованием объявляются подклассами и принимают в качестве аргументов методы, предназначенные для отслеживания истории изменений. Система коллекций была переработана в версии 0.4 с целью расширения ее возможностей в плане использования любых аналогичных коллекциям объектов.

Контроль транзакций

Объект `Session` при обычном сценарии использования поддерживает открытую транзакцию для выполнения всех операций, которая завершается в момент вызова метода `commit` или `rollback`. Объект `SessionTransaction` поддерживает набор объектов `Connection`, который может быть как пустым, так и заполненным, причем каждый объект в нем представляет открытую транзакцию для определенной базы данных. Объект `SessionTransaction` является объектом с отложенной инициализацией, которая начинается при отсутствии данных состояния базы данных. Так как определенная база данных должна участвовать в процессе выполнения запроса, соответствующий этой базе данных объект соединения `Connection` добавляется в список соединений объекта `SessionTransaction`. Хотя обычно в каждый момент времени используется одно соединение с базой данных, поддерживается сценарий использования множества соединений, в котором определенное соединение используется для выполнения определенной операции, в соответствии с ассоциированными с объектами `Table`, `Mapper` данными конфигурации, либо в соответствии с конструкциями языка SQL, применяемыми в рамках операции. При использовании множества соединений также может координироваться процесс выполнения транзакции при применении двухфазной схемы в тех случаях, когда реализация DBAPI предоставляет ее.

20.9. Рабочая единица

Метод `flush` объекта `Session` реализован в рамках отдельного модуля с именем `unitofwork`. Как упоминалось ранее, процесс сохранения данных, скорее всего, является наиболее сложной функцией, реализованной в SQLAlchemy.

Задачей рабочей единицы является перемещение данных из всех ожидающих обработки объектов, присутствующих в коллекции определенного объекта `Session` в базу данных с очисткой коллекций новых (`new`), устаревших (`dirty`) и удаленных (`deleted`) объектов, обрабатываемых объектом `Session`. После завершения этой работы находящиеся в памяти данные состояния объекта `Session` и данные текущей транзакции будут совпадать. Основной трудностью является установление корректной последовательности операций сохранения данных и последующее их выполнение в нужном порядке. Эта задача включает в себя создание списка запросов `INSERT`, `UPDATE` и `DELETE`, включая те запросы, которые были созданы в результате выполнения каскада операций, направленных на удаление или перемещение соответствующих строк; проверку того, что запросы `UPDATE` содержат только те столбцы, которые были действительно изменены; выполнение операций "синхронизации", в ходе которых будут скопированы данные состояния столбцов с первичными ключами в столбцы ссылающимися на них внешними ключами в момент, когда заново генерированные идентификаторы в форме первичных ключей станут доступны; проверку того, что запросы `INSERT` используются в том же порядке, в каком объекты с ними были добавлены в коллекцию объекта `Session`, причем они должны использоваться настолько эффективно, насколько это возможно; а также проверку того, что запросы `UPDATE` и `DELETE` используются в корректном порядке для сокращения вероятности блокировок.

История

Реализация рабочей единицы была начата в форме запутанной системы из структур, возможности которой расширялись бессистемно в каждом отдельном случае; процесс ее разработки может сравниваться с процессом поиска выхода из леса без карты. Ранние ошибки и недостатки функций устраивались путем внесения специфических исправлений и, несмотря на улучшение ситуации после нескольких рефакторингов до версии 0.5, в версии 0.6 модуль рабочей единицы со стабилизованным, хорошо изученным и к тому времени снабженным сотнями тестов кодом должен был быть полностью переработан. После многих недель формирования нового подхода, в рамках которого должны были быть описаны стандартные структуры данных, сам процесс переписывания кода для использования этой новой модели занял всего несколько дней, так как к тому времени идея новой модели была понятна разработчикам. Фактически процесс разработки был значительно упрощен благодаря тому, что принцип работы новой реализации должен был тщательно сопоставляться и приводиться к принципу работы существующей версии. Этот процесс продемонстрировал, что несмотря на то, насколько первая реализация чего-либо является непродуманной, она все же очень ценна, так как является рабочей моделью. Кроме того, он демонстрирует, что полная переработка подсистемы не только допустима, но и является неотъемлемой частью процесса разработки сложных в реализации программных компонентов.

Топологическая сортировка

Ключевой парадигмой, использованной при создании рабочей единицы, является формирование полного списка действий для последующего выполнения в рамках структуры данных, причем каждый элемент этого списка будет представлять отдельный шаг; в области шаблонов проектирования этому подходу соответствует шаблон команд (*command pattern*). Позднее серии "команд" в рамках этой структуры данных располагаются в специфической последовательности с помощью топологической сортировки (*topological sort*). Топологическая сортировка представляет собой процесс, в ходе которого происходит сортировка элементов списка путем их частичного упорядочивания (*partial ordering*), т.е., в этом случае только определенные элементы списка должны быть расположены перед остальными. [Рисунок 20.14](#) иллюстрирует описанный процесс топологической сортировки.

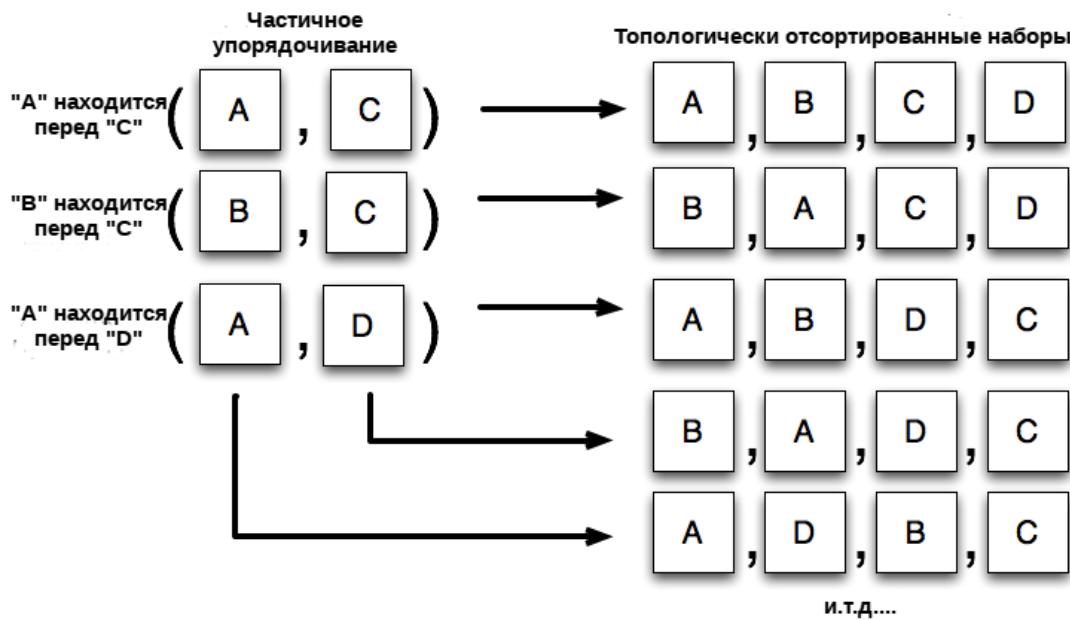


Рисунок 20.14: Топологическая сортировка

Рабочая единица выполняет частичное упорядочивание тех команд сохранения данных, которые должны предшествовать всем остальным. После того, как команды топологически отсортированы, они будут по очереди выполнены. Определение того, какие команды должны предшествовать другим командам, в основном осуществляется путем обнаружения результата выполнения функции *relationship*, которая связывает два объекта Mapper - в общем случае один объект Mapper рас-

сматривается как зависящий от другого объекта, так как функция `relationship` устанавливает зависимость одного объекта `Mapper` от внешнего ключа другого объекта. Существуют похожие правила для установления таблиц соответствия между множествами объектов с обоих сторон, но в данном случае мы будем рассматривать случай использования отношений один ко многим/многие к одному. Зависимости от внешних ключей разрешаются последовательно для предотвращения появления нарушений в области ограничений без необходимости присвоения ограничениям метки "отложенное". Но, что не менее важно, сортировка позволяет использовать первичные ключи, которые генерируются на многих платформах только непосредственно в ходе выполнения запроса `INSERT`, путем извлечения их из набора результирующих данных только что выполненного запроса `INSERT` и вставки в список параметров зависимого запроса для добавления строки. В случае удаления строк данная сортировка производится в обратном порядке - зависимые строки удаляются до того, как происходит удаление строк, от которых они зависят, так как эти строки не будут доступны без наличия внешних ключей, ссылающихся на них.

Рабочая единица реализует систему, в рамках которой топологическая сортировка выполняется на двух различных уровнях, выделяемых на основе структуры имеющихся зависимостей. На первом уровне шаги сохранения данных распределяются между "корзинами" на основе зависимостей между объектами отображения таким образом, что полные "корзины" объектов соответствуют определенному классу. На втором уровне вообще не происходит разделения или происходит разделение одной или нескольких "корзин" на небольшие последовательности объектов с целью обработки случаев использования циклических ссылок или ссылающихся самих на себя таблиц. [Рисунок 20.15](#) иллюстрирует "корзины", сгенерированные для вставки набора объектов `User`, с последующей вставкой набора объектов `Address`, причем на промежуточном шаге должно быть осуществлено копирование только что сгенерированных значений первичных ключей для объектов `User` в столбец внешних ключей с именем `user_id` для каждого из объектов `Address`.

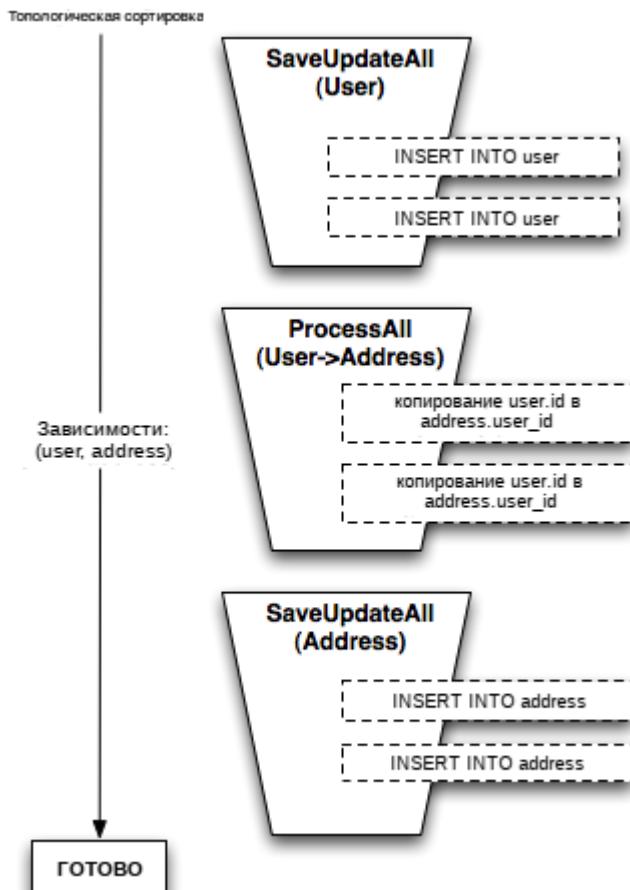


Рисунок 20.15: Организация объектов на основе объектов отображения

В ситуации, когда используется сортировка на основе объектов отображения, любое количество объектов `User` и `Address` может быть сохранено без увеличения сложности шагов или изменения количества "зависимостей", которые должны быть рассмотрены.

На втором уровне сортировки шаги сохранения данных организуются на основе прямых зависимостей между отдельными объектами в рамках области действия одного объекта отображения. Простейшим примером возникновения такой ситуации является таблица, которая содержит внешний ключ, ссылающийся на себя; а также определенная строка, которая должна быть вставлена в таблицу перед другой строкой, ссылающейся на эту строку. Другим примером является набор таблиц с циклическими ссылками (*reference cycle*): таблица A ссылается на таблицу B, которая в свою очередь ссылается на таблицу A. В этом случае некоторые объекты таблицы A должны быть вставлены перед другими объектами для того, чтобы также имелась возможность вставки объектов в таблицы B и C. Ссылающаяся сама на себя таблица является частным случаем циклической ссылки.

Для установления того, какие операции могут оставаться в сформированных на основе объектов `Mapper` корзинах, а также того, какие корзины будут разделены на большие наборы команд для объектов, по отношению к набору зависимостей между объектами отображения применяется алгоритм определения циклических зависимостей, который является модифицированной версией алгоритма определения циклических зависимостей из [блога Guido Van Rossum](#). Те корзины, в которых обнаружены циклические зависимости, впоследствии разделяются на операции, выполняемые по отношению к объектам, и добавляются в коллекцию разделенных на основе объектов отображения корзин путем добавления новых зависимостей от объектов из корзин, разделенных на основе объектов, в объекты из корзин, разделенных на основе объектов отображения. [Рисунок 20.16](#) иллюстрирует процесс разделения корзины для объектов `User` на отдельные команды объектов путем использования функции `relationship` для указания на зависимость объекта `User` от своего атрибута `contact`.

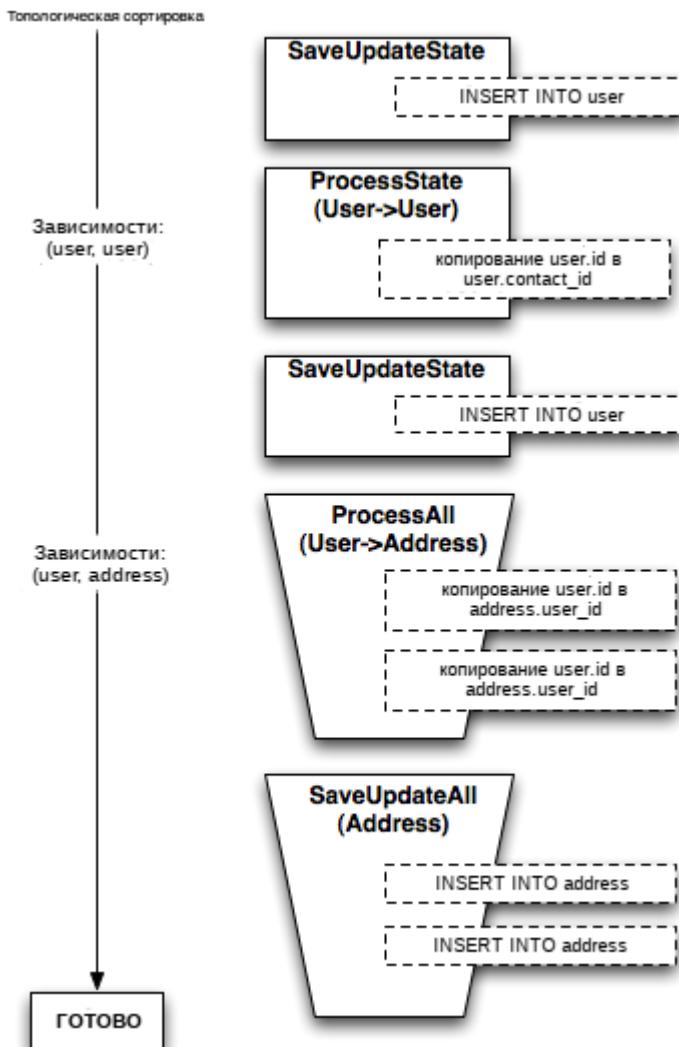


Рисунок 20.16: Преобразование циклических ссылок в отдельные шаги процесса

Обоснованием применения структуры корзин является возможность упорядочивания стандартных запросов настолько, насколько это возможно, путем сокращения числа шагов, выполняемых с использованием функций языка Python, а также возможность осуществления более эффективных взаимодействий с реализацией DBAPI, которые иногда позволяют выполнить тысячи запросов в рамках вызова одного метода на уровне языка Python. Более сложный метод определения индивидуальных зависимостей для объектов используется лишь в случае существования циклических ссылок между объектами отображения, но этот метод используется исключительно в тех частях графа объектов, где он требуется.

20.10. Заключение

С самого начала работы над SQLAlchemy перед разработчиками были поставлены значительные задачи, причем главной целью разработки было создание программного продукта для работы с базами данных, обладающего настолько большим набором возможностей и являющегося настолько гибким, насколько это возможно. Это задача была выполнена в ходе работы над поддержкой реляционных баз данных, так как было понятно, что всеобъемлющая и проработанная поддержка реляционных баз данных является основной задачей; и даже сейчас масштабы данной работы являются гораздо большими, чем казалось ранее.

Основывающийся на компонентах подход был предназначен для извлечения возможной пользы из каждой области возможностей путем предоставления множества различных элементов, которые

приложения могли использовать по отдельности или комбинировать. Эту систему было интересно создавать, поддерживать и внедрять.

Для разработки был осознанно выбран медленный курс, основывающийся на предположении о том, что методичная, всесторонняя проработка основных функций в конечном итоге окажется более удачной, нежели быстрая реализация функций без должной проработки. Потребовалось много времени для того, чтобы система SQLAlchemy стала последовательной и хорошо документированной с точки зрения пользователя, но в ходе процесса разработки архитектура проекта всегда была на шаг впереди, что в некоторых случаях приводило к проявлению эффекта "машины времени", при котором функции могли добавляться до того, как пользователи запрашивали их.

Язык программирования Python был надежной основой (если быть немного привередливым, то можно отметить, в частности, область производительности). Последовательность и в значительной степени открытая модель времени выполнения позволили лучше реализовать в рамках SQLAlchemy те возможности, которые предоставлялись аналогичными программными продуктами, разработанными с использованием других языков программирования.

Участники проекта SQLAlchemy надеются, что язык программирования Python получит еще большее распространение в таком широком спектре областей и на таком большом количестве предприятий, как это возможно, а также на то, что масштабы использования реляционных баз данных останутся значительными и будут расширяться. Целью проекта SQLAlchemy является демонстрация того, что реляционные базы данных, язык программирования Python и хорошо продуманные объектные модели являются очень ценными инструментами разработчиков.

21. Twisted

Глава 21 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Twisted является управляемым событиями сетевым фреймворком, разработанным с использованием языка программирования Python. Он был создан в начале 2000 годов, когда в распоряжении разработчиков сетевых игр было несколько масштабируемых и не являющихся кроссплатформенными библиотек для работы с сетью, предназначенных для различных языков программирования. Авторы Twisted, пытаясь разрабатывать игры с использованием существующих программных компонентов для работы с сетью, боролись с их недостатками и, четко ощущая необходимость в масштабируемом управляемом событиями кроссплатформенном сетевом фреймворке, решили разработать такой программный компонент, извлекая уроки из ошибок и трудностей, с которыми сталкивались разработчики сетевых игр и приложений в прошлом.

Twisted поддерживает большое количество стандартных протоколов как транспортного, так и прикладного уровня, включая TCP, UDP, SSL/TLS, HTTP, IMAP, SSH, IRC и FTP. Как и в случае языка, с использованием которого он разработан, все необходимые компоненты присутствуют в его комплекте поставки изначально (такой подход в рамках языка Python называется "batteries-included"); в составе Twisted присутствуют реализации серверов и клиентов для всех поддерживаемых протоколов, а также утилиты, упрощающие процесс настройки и развертывания приложений на основе Twisted промышленного уровня с использованием командной строки.

21.1. Почему Twisted?

В 2000 году человек по прозвищу glyrh, являющийся создателем фреймворка Twisted, работал над текстовой многопользовательской игрой с названием "Twisted Reality" ("Запутанная реальность"). Она была реализована с помощью нагромождения потоков, по 3 потока на каждое соединение, причем для разработки был выбран язык Java. В рамках соединения присутствовал поток ввода, блокирующийся операциями чтения, поток вывода, который должен был блокироваться опреде-

ленным типом операций записи, а также поток "обработки логики", который должен был бездействовать, ожидая истечения времени таймеров или поступления событий в очередь. По мере передвижения игроков по виртуальному пространству и их взаимодействия, происходили взаимные блокировки потоков, портились данные в кэшах и логика блокировок никогда не была достаточно хорошо проработана - использование потоков делало программное обеспечение запутанным, наполненным ошибками и трудно масштабируемым.

В поисках альтернатив, он столкнулся с языком программирования Python, а именно, с модулем Python `select`, предназначенный для мультиплексирования операций ввода/вывода таких объектов, работающих с потоками данных, как сокеты и программные каналы (спецификация Single UNIX Specification, Version 3 (SUSv3) описывает функцию API для вызова `select`). В то время в рамках языка Java не осуществлялось раскрытие интерфейса `select` операционной системы, как и любых других API для осуществления операций асинхронного ввода/вывода (пакет `java.nio` для неблокирующих операций ввода/вывода был добавлен в комплект поставки J2SE 1.4, причем эта версия была выпущена в 2002 году). Начальный прототип игры, созданный с использованием языка Python и метода `select`, сразу же послужил доказательством меньшей запутанности кода и большей надежности по сравнению с версией на основе потоков.

В ходе преобразования кода для использования языка Python, метода `select` и парадигмы управления событиями, `glyph` разработал клиент и сервер игры на языке Python с использованием API `select`. Но после этого ему захотелось большего. В качестве фундаментальной идеи он рассматривал возможность перевода операций взаимодействия с сетью в рамки вызовов методов объектов игры. А что, если вы сможете получать сообщения электронной почты в игре, также, как это реализовано в почтовом демоне Nethack? А если у каждого игрока будет возможность создать домашнюю страницу? `Glyph` понял, что ему необходимы хорошие клиенты и серверы для протоколов IMAP и HTTP, разработанные с использованием языка программирования Python и метода `select`.

Изначально он использовал платформу [Medusa](#), созданную в середине 90 годов для разработки сетевых серверов на языке Python и основанную на модуле `asyncore`. Модуль `asyncore` реализует механизм работы с сокетом, в рамках которого поверх API `select` операционной системы создается интерфейс, состоящий из диспетчера и функции обратного вызова.

Эта находка вдохновила разработчика `glyph`, но платформа `Medusa` имела два недостатка:

1. В 2001 году, когда `gliph` начал работу над Twisted Reality, ее развитие практически не поддерживалось.
2. Модуль `asyncore` был такой тонкой прослойкой над функциями для работы с сокетами, что разработчикам приложений все еще приходилось напрямую производить манипуляции с ними. Это означало, что о портируемости кода должен был заботиться разработчик. К тому же, в то время поддержка платформы Windows в рамках модуля `asyncore` была реализована с ошибками, а `gliph` был уверен в том, что ему было необходимо иметь возможность запускать графический интерфейс клиента на платформе Windows.

`Glyph` столкнулся с перспективой самостоятельной реализации платформы для работы с сетью и решил, что игра Twisted Reality выявила проблему, решение которой было не менее интересным, чем разработка самой игры.

Со временем игра Twisted Reality превратилась в платформу для работы с сетью Twisted, которая могла выполнять действия, не доступные для других сетевых платформ языка Python:

- Использовать парадигму управляемого событиями программирования вместо многопоточного программирования.
- Быть кросплатформенной: предоставлять унифицированный интерфейс для систем уведомления о наступлении событий, раскрываемый во всех широко используемых операционных системах.
- Включать все компоненты в комплект поставки (реализовывать подход "batteries-included"): предоставлять реализации классов для работы с популярными протоколами прикладного уровня сразу после установки, таким образом платформа Twisted без лишних действий становится полезной для разработчиков.
- Соответствовать стандартам RFC и подтверждать это соответствие с помощью мощного набора тестов.
- Быть удобной для совместного использования множества сетевых протоколов.

- Иметь возможность расширения функций.

21.2. Архитектура Twisted

Twisted является управляемым событиями сетевым фреймворком. Парадигма управляемого событиями программирования является настолько неотъемлемой частью философии проектирования Twisted, что стоит уделить некоторое время обзору того, что на самом деле понимают под парадигмой управляемого событиями программирования.

Управляемое событиями программирование осуществляется в соответствии с парадигмой, определяющей то, что ход выполнения программы задается внешними событиями. Для него характерно использование цикла приема событий и использование функций обратного вызова с целью выполнения действий в моменты наступления событий. Другим двумя стандартными парадигмами программирования являются синхронное (однопоточное) и многопоточное программирование.

Давайте сравним и найдем отличия между однопоточной, многопоточной и управляемой событиями моделями программирования с помощью примера. На [Рисунке 21.1](#) представлена выполняемая программой работа с течением времени в случае использования каждой из этих трех моделей. Программа должна выполнить три задачи, причем каждая из этих задач блокирует выполнение программы до момента завершения операции ввода/вывода. Время, потраченное при блокировании выполнения программы для завершения операций ввода/вывода, показано с помощью областей серого цвета.

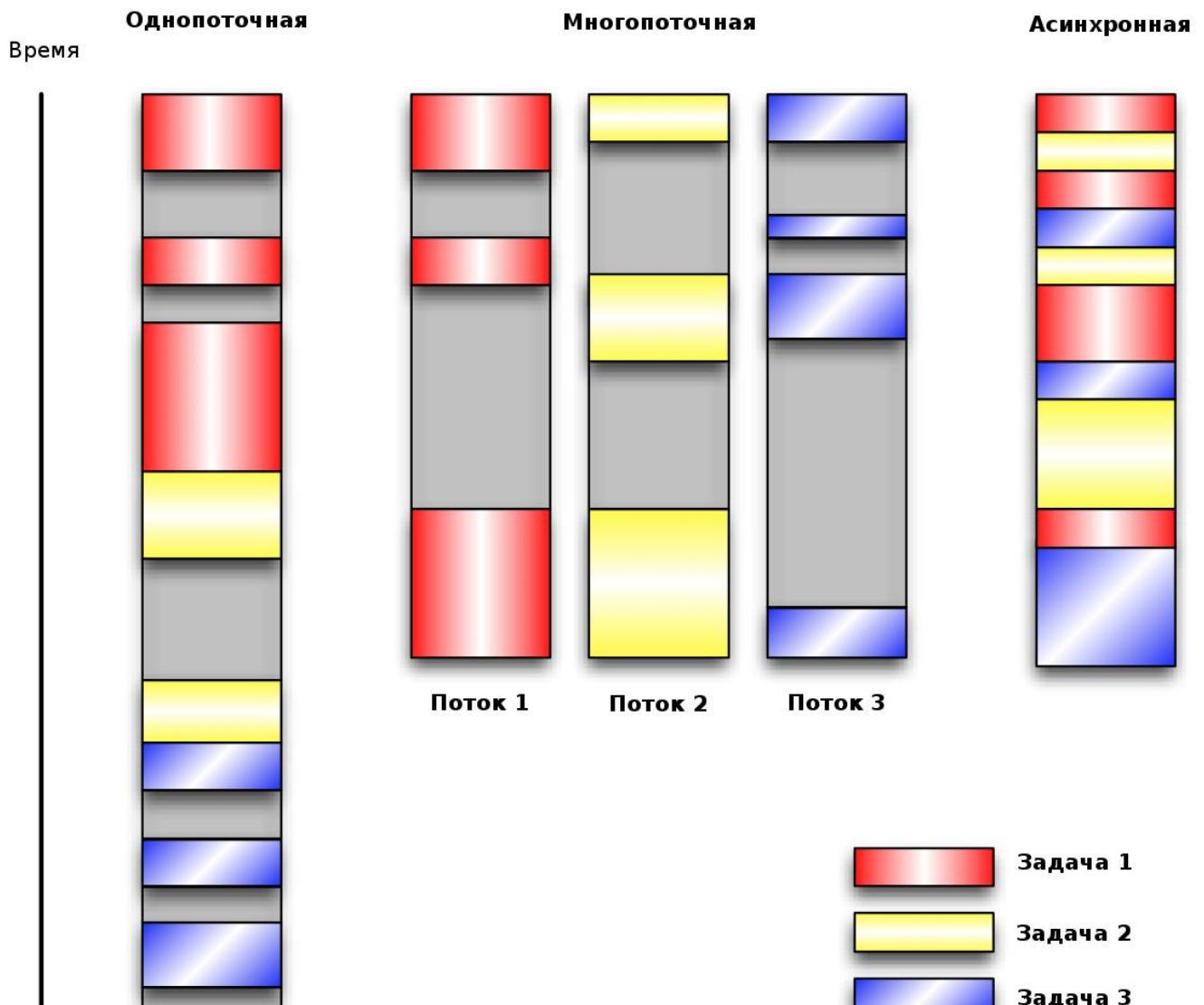


Рисунок 21.1: Модели сетевого программирования

В однопоточной синхронной версии программы задачи выполняются по очереди. В том случае, если одна задача блокируется для выполнения операций ввода-вывода, всем другим задачам приходится ожидать окончания выполнения операций и приступать к работе по очереди. Этот определенный порядок и последовательное выполнение задач упрощают понимание принципа работы программы, но при этом программа становится чрезмерно медленной в том случае, когда задачи не зависят друг от друга, но им все равно приходится ожидать друг друга.

В многопоточной версии программы три задачи с блокировками во время выполнения работы исполняются в отдельных программных потоках. Эти потоки управляются операционной системой и могут исполняться одновременно на множестве процессоров или поочередно на одном процессоре. Это позволяет выполнять работу некоторым потокам в тот момент, когда другие потоки блокируются для обработки ресурсов. Обычно эта программа более оптимальна с точки зрения времени, чем аналогичная синхронная программа, но приходится разрабатывать дополнительный код для защиты разделяемых потоками ресурсов, доступ к которым может осуществляться одновременно из нескольких потоков. Принцип работы многопоточных программ может оказаться более сложным для понимания, так как в случае реализации таких программ приходится заботиться о

безопасной работе потоков с помощью механизмов сериализации процессов (блокировок), использования реентерантных функций, локальных хранилищ данных для потоков или других механизмов, которые в случае некорректного применения могут приводить к скрытым и опасным ошибкам.

Управляемая событиями версия программы поочередно выполняет три задачи, но в рамках одного потока. При выполнении операций ввода/вывода или других затрачивающих время операций, в цикле обработки событий регистрируется функция обратного вызова и выполнение задачи продолжается после завершения операции ввода/вывода. Функция обратного вызова устанавливает метод обработки события сразу же после завершения операции. Цикл обработки событий ожидает события и распределяет их в порядке поступления для обработки с помощью функций обратного вызова, которые их ожидают. Это обстоятельство позволяет программе выполнять работу всегда, когда это возможно без необходимости использования множества дополнительных потоков. Управляемые событиями программы могут быть более простыми для понимания, нежели многопоточные программы, так как разработчику не приходится заботиться о безопасности функционирования потоков.

Управляемая событиями модель обычно является хорошим выбором, если:

1. имеется множество задач, которые...
2. в значительной степени независимы (поэтому им не нужно взаимодействовать друг с другом или ожидать друг друга), а также...
3. некоторые из этих задач блокируются во время ожидания событий.

Также эта модель является хорошим выбором в том случае, когда приложению требуется предоставить задачам доступ к изменяемым данным, так как в этом случае не требуется синхронизации.

Приложения для работы с сетью обычно имеют именно эти свойства, что и определяет их хорошую совместимость с управляемой событиями моделью программирования.

Повторное использование существующих приложений

Многие популярные клиенты и серверы для различных сетевых протоколов существовали до того, как был создан фреймворк Twisted. Почему же *glyph* просто не использовал Apache, IRCd, BIND, OpenSSH или любое другое из ранее существующих приложений, клиенты и серверы из которых были с нуля реализованы в рамках Twisted?

Проблема состоит в том, что все эти реализации серверов обычно использовали разработанный с нуля с использованием языка программирования С код для взаимодействия с сетью, причем код приложения был напрямую связан с кодом сетевого уровня. Это обстоятельство в значительной степени затрудняло использование существующего кода в форме библиотек. Эти программные компоненты должны были совместно использоваться как черные ящики, не позволяя разработчику повторно использовать код в том случае, если он или она пожелали передать одни и те же данные с использованием множества протоколов. К тому же, реализации серверов и клиентов обычно являются отдельными приложениями, которые совместно не используют код. Расширение возможностей этих приложений и поддержка кроссплатформенной клиент-серверной совместимости оказываются более сложными задачами, чем должны.

В случае использования Twisted клиенты и серверы разрабатываются с использованием языка Python и постоянного интерфейса. Это обстоятельство упрощает разработку клиентов и серверов, совместное использование кода для реализации клиентов и серверов, использование одной и той же логики приложения для работы с различными протоколами, а также тестирование кода.

Шаблон проектирования reactor

Twisted реализует шаблон проектирования reactor, который описывает возможность демультиплексирования и распределения событий от множества источников между их обработчиками в однопоточном окружении.

Основной частью фреймворка Twisted является цикл обработки событий, спроектированный согласно шаблону reactor. Этот цикл располагает информацией о событиях сети, файловой системы и таймеров. Он ожидает и при наступлении обрабатывает эти события, абстрагируясь от специфичного для платформы поведения и представляя интерфейсы для осуществления ответа на события в любой точке сетевого стека без сложностей.

Цикл обработки событий в основном выполняет следующие действия:

```
while True:
    timeout = time_until_next_timed_event()
    events = wait_for_events(timeout)
    events += timed_events_until(now())
    for event in events:
        event.process()
```

Цикл обработки событий основывается на API `poll` (описанном в спецификации Single UNIX Specification, Version 3 (SUSv3)), используемом в качестве стандарта для всех платформ. Дополнительно Twisted поддерживает несколько платформо-специфичных гибко масштабируемых API для мультиплексирования. Платформо-специфичные циклы обработки событий могут основываться на KQueue, методе мультиплексирования, на основе механизма `kqueue` из состава FreeBSD, `epoll` в системах с поддержкой интерфейса `epoll` (на данный момент это Linux 2.6) и IOCP на основе технологии Windows Input/Output Completion Ports.

Примеры зависящих от реализации особенностей процесса ожидания событий, рассматриваемых Twisted, включают следующие:

- Ограничения сети и файловой системы.
- Особенности буферизации.
- Метод установления факта разрыва соединения.
- Данные, возвращаемые в случаях ошибок.

Реализация циклов обработки событий в рамках Twisted также заботится о корректном использовании низкоуровневых не использующих блокировок API, а также о корректной работе в случае нестандартных критических ситуаций. В рамках языка Python API IOCP не раскрывается вообще, поэтому Twisted использует свою собственную реализацию.

Управление цепочками функций обратного вызова

Функции обратного вызова являются фундаментальной частью процесса разработки управляемых событиями систем и способом указания со стороны цикла обработки событий на факт наступления событий. По мере роста управляемых событиями программ, обработка и удачных и неудачных вариантов событий в рамках приложения значительно усложняется. Невозможность регистрации подходящей функции обратного вызова может привести к блокировке программы на операции обработки события, которая никогда не будет выполнена, при этом ошибки могут распространяться по цепочке функций обратного вызова от сетевого стека через уровни приложения.

Давайте рассмотрим некоторые ловушки, появляющиеся в управляемых событиями программах, в ходе сравнения синхронной и асинхронной версии игрушечной утилиты для получения страницы на основе строки URL, разработанной с использованием похожего на Python псевдокода:

Синхронная версия программы для получения страницы на основе строки URL:

```

import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Завершение работы..."
    exit(0)

url = "http://google.com"
try:
    page = getPage(url)
    processPage(page)
except Error, e:
    logError(error)
finally:
    finishProcessing()

```

Асинхронная версия программы для получения страницы на основе строки URL:

```

from twisted.internet import reactor
import getPage

def processPage(page):
    print page
    finishProcessing()

def logError(error):
    print error
    finishProcessing()

def finishProcessing(value):
    print "Завершение работы..."
    reactor.stop()

url = "http://google.com"
# Функция getPage принимает следующие аргументы: строку url,
# функцию обратного вызова для использования в случае удачного получения страницы,
# функцию обратного вызова для обработки ошибки
getPage(url, processPage, logError)

reactor.run()

```

В случае асинхронной программы для получения страницы на основе URL вызов `reactor.run()` запускает цикл обработки сообщений. И в синхронной, и в асинхронной версиях гипотетическая функция `getPage` выполняет работу по получению страницы. Функция `processPage` вызывается в случае успешного получения страницы, а функция `logError` вызывается в случае возникновения исключения (`Exception`) при попытке получения страницы. В любом случае после этого вызывается функция `finishProcessing`.

Вызов функции `logError` из асинхронной версии заменяется на часть `except` блока `try/except` в синхронной версии. Вызов функции `processPage` заменяет оператор `else` и безусловный вызов функции `finishProcessing` заменяет оператор `finally`.

В синхронной версии с помощью структуры блока `try/except`, вызывается только одна из функций `logError` и `processPage` и функция `finishProcessing` всегда вызывается единожды; в асинхронной версии разработчик ответственен за использование корректных цепочек функций обрат-

ного вызова для обработки удачного или неудачного завершения процесса. Если бы в случае ошибки программирования вызов функции `finishProcessing` не использовал функции `processPage` и `logError` вместе с соответствующими цепочками функций обратных вызовов, цикл обработки событий не прервал бы никогда свою работу и программа выполнялась бы вечно.

Этот простейший пример указывает на сложности, которые расстраивали разработчиков Twisted в течение первых лет работы над проектом. Ответом на эти сложности послужило увеличение размеров объекта с названием `Deferred`.

Объекты `Deferred`

Объект `Deferred` является абстракцией над результатом, которого на данный момент не существует. Он также облегчает управление цепочками функций обратного вызова, используемыми для получения этого результата. При возврате из функции объект `Deferred` выступает в роли обещания того, что функция вернет результат на определенном шаге. Этот возвращенный объект `Deferred` содержит ссылки на все функции обратного вызова, зарегистрированные для данного события, поэтому только этот единственный объект должен передаваться между функциями и гораздо проще работать с ним, а не управлять отдельными функциями обратного вызова.

Объектам `Deferred` соответствуют пары цепочек функций обратных вызовов, одна для обработки успешных операций (с помощью функций обратного вызова для обработки данных) и одна для обработки ошибок (с помощью специальных функций обратного вызова для обработки ошибок). Объекты `Deferred` создаются с двумя изначально пустыми цепочками. После этого добавляются пары функций обратного вызова для обработки данных и ошибок, которые будут использоваться для обработки удачных и неудачных операций в каждой точке процесса обработки событий. В тот момент, когда осуществляется асинхронная доставка результата выполнения операции, "активизируется" объект `Deferred` и вызываются соответствующие функции обратного вызова для обработки данных и событий в том порядке, в котором они были добавлены.

Ниже приведена использующая объекты `Deferred` версия асинхронной программы для получения страницы на основе URL в псевдокоде:

```
from twisted.internet import reactor
import getPage

def processPage(page):
    print page

def logError(error):
    print error

def finishProcessing(value):
    print "Завершение работы..."
    reactor.stop()

url = "http://google.com"
deferred = getPage(url) # Функция getPage возвращает объект Deferred
deferred.addCallbacks(success, failure)
deferred.addBoth(stop)

reactor.run()
```

В этой версии вызываются те же обработчики событий, но все они регистрируются с помощью одного объекта `Deferred` вместо распределения по коду и передачи имен функций в качестве аргументов функции `getPage`.

Объект Deferred создается с двумя уровнями функций обратного вызова. Во-первых, функция addCallbacks добавляет функцию обработки данных processPage и функцию обработки ошибок logError на первый уровень их соответствующих цепочек. После этого с помощью функции addBoth функция обратного вызова finishProcessing добавляется на второй уровень обеих цепочек.

Цепочки функций обратного вызова в форме диаграммы выглядят примерно так, как показано на [Рисунке 21.1](#).

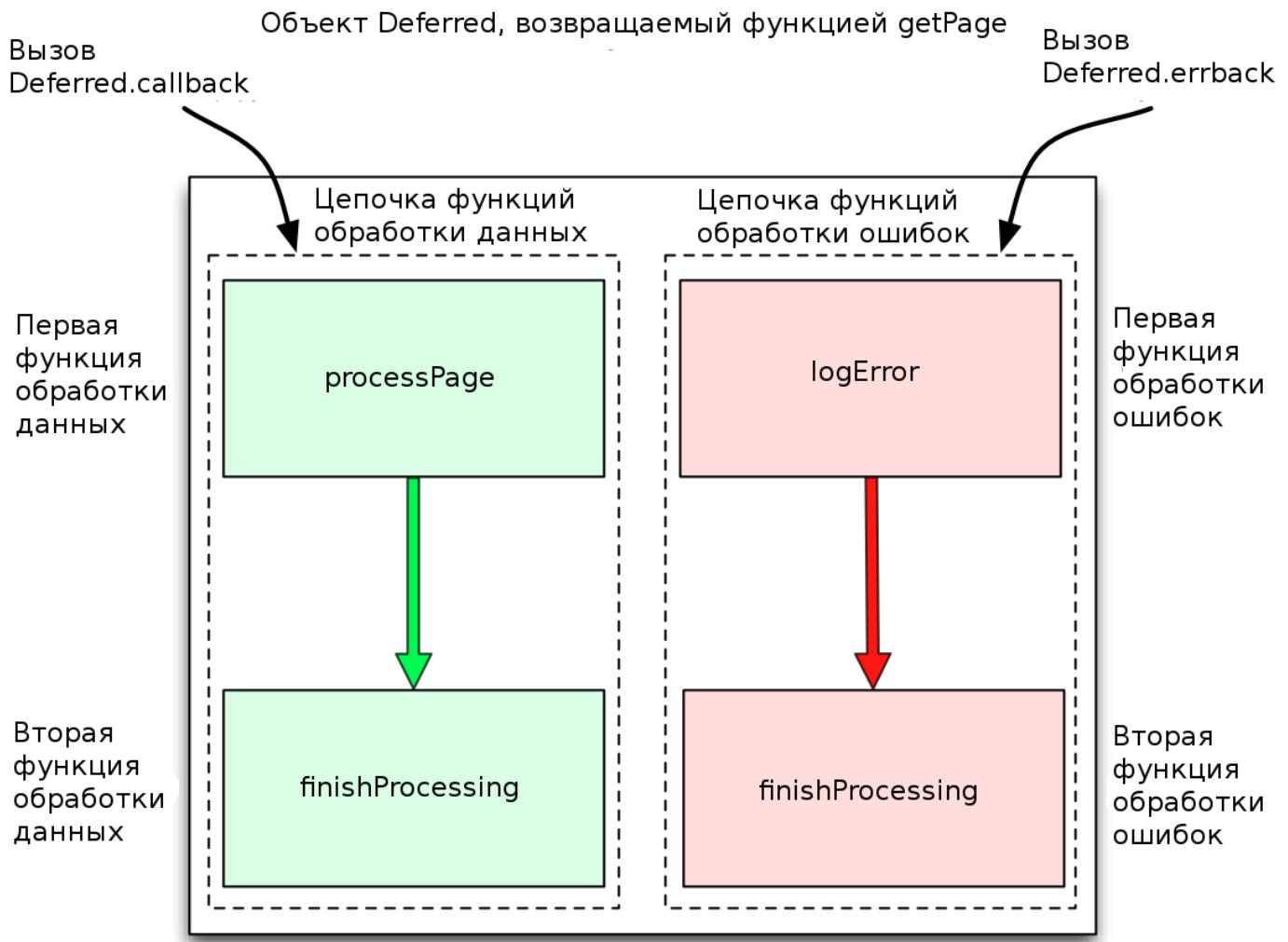


Рисунок 21.1: Цепочки функций обратного вызова

Объекты Deferred могут быть активизированы только однократно; повторная попытка их активизации приведет к возникновению исключения (Exception). Это позволяет добиться семантики объектов Deferred, аналогичной семантике блоков try/except в синхронных версиях, которые упрощают понимание процесса обработки асинхронных событий и позволяют избежать коварных ошибок, вызванных тем, что функции обратного вызова вызываются больше одного раза для события.

Понимание принципов использования объектов Deferred является важным для понимания принципа работы программ на основе Twisted. Однако, при использовании высокоуровневых абстракций, предоставляемых фреймворком Twisted для сетевых протоколов, обычно вообще не приходится использовать объекты Deferred напрямую.

Абстракция `Deferred` является достаточно мощной и была заимствована другими управляемыми событиями платформами, в числе которых jQuery, Dojo и Mochkit.

Транспорты

Транспорты представляют соединение между двумя конечными точками, взаимодействующими посредством сети. Транспорты отвечают за описание параметров соединения, таких, как является ли соединение потоко- или дейтаграммно-ориентированным, какие алгоритмы применяются для контроля потока, а также какова надежность соединения. TCP-, UDP- и Unix-сокеты являются примерами транспортов. Они спроектированы так, чтобы быть "минимально функциональными примитивами, которые могут быть максимально используемыми повторно" и отделены от реализаций протоколов, позволяя множеству протоколов использовать один и тот же тип транспорта. Транспорты реализуют интерфейс `ITransport`, который имеет следующие методы:

<code>write</code>	Записать последовательно какие-либо данные в физическое соединение без блокировок.
<code>writeSequence</code>	Записать список строк в физическое соединение.
<code>loseConnection</code>	Записать все ожидающие данные, после чего закрыть соединение.
<code>getPeer</code>	Получить удаленный адрес данного соединения.
<code>getHost</code>	Получить локальный адрес данного соединения.

Отделение транспортов от протоколов также упрощает тестирование двух уровней. Исследуемый транспорт может просто записать данные в строку для проверки.

Протоколы

Протоколы устанавливают методы асинхронной обработки событий сети, HTTP, DNS и IMAP являются примерами прикладных сетевых протоколов. Протоколы реализуют интерфейс `IProtocol`, который имеет следующие методы:

<code>makeConnection</code>	Создать соединение для транспорта с сервером.
<code>connectionMode</code>	Вызывается тогда, когда соединение создано.
<code>dataReceived</code>	Вызывается тогда, когда приняты данные.
<code>connectionLost</code>	Вызывается при закрытии соединения.

Связь между циклом обработки событий, протоколами и транспортами может быть лучшим образом проиллюстрирована с помощью примера. Ниже приведены завершенные реализации эхосервера и клиента, сначала код сервера:

```
from twisted.internet import protocol, reactor

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        # Любые данные после приема должны быть отправлены назад
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000, EchoFactory())
reactor.run()
```

И код клиента:

```

from twisted.internet import reactor, protocol

class EchoClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("hello, world!")

    def dataReceived(self, data):
        print "Ответ сервера:", data
        self.transport.loseConnection()

    def connectionLost(self, reason):
        print "соединение разорвано"

class EchoFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return EchoClient()

    def clientConnectionFailed(self, connector, reason):
        print "Не удалось осуществить соединение - всего доброго!"
        reactor.stop()

    def clientConnectionLost(self, connector, reason):
        print "Соединение разорвано - всего доброго!"
        reactor.stop()

reactor.connectTCP("localhost", 8000, EchoFactory())
reactor.run()

```

В ходе выполнения сценария сервер должен начать работу TCP-сервер, ожидающий соединений на порту 8000. Сервер использует протокол Echo и данные передаются с помощью транспорта TCP. В ходе выполнения сценария клиента с сервером организуется соединение по протоколу TCP, после чего серверу передаются данные, которые он присыпает назад, соединение закрывается, а цикл обработки событий завершает свою работу. Фабрики протоколов (EchoFactory) используются для создания экземпляров классов протоколов на двух сторонах соединения. С двух сторон соединения производится асинхронный обмен данными; метод connectTCP осуществляет регистрацию функций обратного вызова в цикле обработки событий для получения уведомлений о том, что данные доступны для чтения из сокета.

Приложения

Twisted является фреймворком для создания масштабируемых, кроссплатформенных сетевых серверов и клиентов. Упрощение процесса развертывания этих приложений в соответствии со стандартом в промышленных окружениях является важным шагом для подобной платформы, направленным на расширение ее распространения.

Для этого в рамках проекта Twisted была разработана инфраструктура приложений Twisted, предоставляющая универсальный и конфигурируемый способ развертывания приложения на основе Twisted. Она позволяет разработчику избежать использования шаблонного кода, интегрируя приложение с существующими инструментами для изменения принципа его работы, что подразумевает возможности запуска приложения в режиме демона, записи данных событий в журнал, использования измененных циклов обработки событий, использования функций профилирования, а также внедрение других возможностей.

Инфраструктура приложения состоит из четырех основных частей: Служб (Services), Приложений (Applications), системы управления конфигурацией (с помощью ТАС-файлов и плагинов) и утилиты командной строки twistd. Для иллюстрации этой инфраструктуры мы превратим эхо-сервер из предыдущего раздела в приложение.

Служба

Под службой понимается все то, что может быть запущено и остановлено, а также создается с использованием интерфейса `IService`. В комплекте поставки Twisted присутствуют реализации служб для TCP, FTP, HTTP, SSH, DNS и многих других протоколов. Множество служб может регистрироваться одним приложением.

Основой интерфейса `IService` служат следующие методы:

<code>startService</code>	Запуск службы. Этот процесс может включать стадии загрузки данных конфигурации, установления соединений с базой данных или начала ожидания соединений на порту.
<code>stopService</code>	Остановка службы. Этот процесс может включать стадии сохранения данных на диск, закрытия соединений с базой данных или окончания приема соединений на порту.

Наша эхо-служба использует протокол TCP, поэтому мы можем использовать стандартную реализацию интерфейса `IService` с названием `TCPServer`.

Приложение

Приложение является высокоуровневой службой, представляющей всю программу, созданную с использованием Twisted. Службы регистрируют себя в рамках приложений и описанная выше утилита развертывания `twistd` ищет и запускает приложения.

Мы создадим эхо-приложение, в рамках которого может быть зарегистрирована эхо-служба.

Файлы TAC

При работе с приложениями Twisted, реализованными с использованием обычных файлов исходного кода языка Python, разработчик несет ответственность за разработку кода для запуска и остановки цикла обработки событий, а также настройки приложения. В рамках инфраструктуры приложений Twisted реализации протоколов располагаются в модулях, использующие данные протоколы службы регистрируются с помощью файла конфигурации приложения Twisted (Twisted Application Configuration (TAC)), а цикл обработки событий и процесс конфигурации управляются внешней утилитой.

Для того, чтобы превратить наш эхо-сервер в эхо-приложение, мы можем использовать следующий простой алгоритм:

1. Перенести части кода эхо-сервера, относящиеся к реализации протокола, в отдельный, предназначенный для них модуль.
2. В TAC-файле:
 1. Создать эхо-приложение.
 2. Создать экземпляр службы `TCPServer`, который будет использовать наш класс `EchoFactory` и зарегистрировать его в рамках приложения.

Код для управления циклом приема событий с использованием утилиты `twistd` будет обсуждаться ниже. В итоге код приложения будет аналогичен следующему:

Файл `echo.py`:

```
from twisted.internet import protocol, reactor

class Echo(protocol.Protocol):
    def dataReceived(self, data):
        self.transport.write(data)

class EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()
```

Файл echo_server.tac:

```
from twisted.application import internet, service
from echo import EchoFactory

application = service.Application("echo")
echoService = internet.TCPServer(8000, EchoFactory())
echoService.setServiceParent(application)
```

twistd

`twistd` (произносится "твист-ди") является кроссплатформенной утилитой для развертывания приложений, созданных с использованием Twisted. Она исполняет ТАС-файлы и обрабатывает запуск и остановку приложения. Согласно подходу "batteries-included", используемому в ходе разработки сетевого фреймворка Twisted, утилита `twistd` поддерживает множество полезных параметров конфигурации, в том числе запуск приложения в режиме демона, указание расположения файлов журнала, снижение системных привилегий, запуск приложения в окружении chroot, запуск приложения с нестандартным циклом обработки событий или даже запуск приложения под управлением программы для профилирования.

Мы можем запустить наше эхо-приложение с помощью команды:

```
twistd -y echo_server.tac
```

В простейшем случае `twistd` запустит экземпляр приложения в режиме демона, причем запись сообщений о событиях будет осуществляться в файл с именем `twistd.log`. После запуска и остановки приложения файл журнала событий будет содержать подобные строки:

```
2011-11-19 22:23:07-0500 [-] Log opened.
2011-11-19 22:23:07-0500 [-] twistd 11.0.0 (/usr/bin/python 2.7.1) starting up.
2011-11-19 22:23:07-0500 [-] reactor class: twist-
ed.internet.selectreactor.SelectReactor.
2011-11-19 22:23:07-0500 [-] echo.EchoFactory starting on 8000
2011-11-19 22:23:07-0500 [-] Starting factory
2011-11-19 22:23:20-0500 [-] Received SIGTERM, shutting down.
2011-11-19 22:23:20-0500 [-] (TCP Port 8000 Closed)
2011-11-19 22:23:20-0500 [-] Stopping factory
2011-11-19 22:23:20-0500 [-] Main loop terminated.
2011-11-19 22:23:20-0500 [-] Server Shut Down.
```

Использование инфраструктуры приложений Twisted для запуска служб позволяет разработчикам избежать написания шаблонного кода для таких стандартных функций службы, как работа с журналом событий и переход в режим демона. Также она позволяет использовать стандартный интерфейс командной строки для развертывания приложений.

Плагины

Альтернативой системе на основе файлов ТАС, предназначеннной для запуска приложений Twisted, является система плагинов. В то время, как система на основе файлов ТАС упрощает регистрацию простых иерархий заданных в рамках конфигурационного файла приложения служб, система плагинов упрощает регистрацию нестандартных служб, используя для этого подкоманды утилиты `twistd`, а также расширение интерфейса командной строки приложения.

Особенности использования этой системы:

1. Требование стабильности предъявляется только к API плагинов, что упрощает процесс усовершенствования сторонними разработчиками программного обеспечения.
2. В коде предусмотрена возможность поиска плагинов. Плагины могут быть загружены и сохранены при первом запуске программы, повторно найдены при каждом запуске программы или их наличие может проверяться периодически в ходе исполнения программы, что позволяет определять наличие новых плагинов, установленных после запуска программы.

Для расширения возможностей программы с использованием системы плагинов Twisted необходимо просто создать объекты, реализующие интерфейс `IPlugin`, после чего поместить файл с их объявлениями в определенное место, в котором их будет искать система плагинов.

После преобразования нашего эхо-сервера в приложение Twisted, преобразование его в плагин Twisted будет достаточно простой задачей. К созданному ранее модулю `echo`, содержащему описание протокола Echo и объявления `EchoFactory`, мы добавим директорию с именем `twisted`, содержащую поддиректорию с именем `plugins`, которая в свою очередь будет содержать объявления классов для нашего эхо-плагина. Этот плагин позволит нам запустить эхо-сервер и указать используемый порт с помощью аргументов утилиты `twistd`:

```
from zope.interface import implements

from twisted.python import usage
from twisted.plugin import IPlugin
from twisted.application.service import IServiceMaker
from twisted.application import internet

from echo import EchoFactory

class Options(usage.Options):
    optParameters = [["port", "p", 8000, "Номер порта для приема соединений."]]

class EchoServiceMaker(object):
    implements(IServiceMaker, IPlugin)
    tapname = "echo"
    description = "Эхо-сервер на основе протокола TCP."
    options = Options

    def makeService(self, options):
        """
        Создается объект TCPServer с помощью фабрики из модуля проекта.
        """
        return internet.TCPServer(int(options["port"]), EchoFactory())

serviceMaker = EchoServiceMaker()
```

Наш эхо-сервер сейчас будет представлен параметром сервера в выводе команды `twistd --help` и команда `twistd echo --port=1235` позволит запустить эхо-сервер на порту 1235.

В составе Twisted имеется модульная система аутентификации для серверов с названием `twisted.cred` и система плагинов обычно используются для добавления шаблона аутентификации в приложение. Возможно использование `AuthOptionMixed` из состава `twisted.cred` для добавления поддержки стандартных систем аутентификации в приложение для командной строки или для добавления нового типа аутентификации. Например, с помощью системы плагинов может быть добавлена возможность аутентификации с помощью стандартной базы данных паролей Unix или сервера LDAP.

В комплекте поставки `twistd` находятся плагины для множества поддерживаемых Twisted протоколов, что превращает работу по созданию сервера в работу по вводу одной команды. Ниже приведено несколько примеров серверов `twistd`, поставляемых в составе Twisted:

```
twistd web --port 8080 --path .
```

```

Запуск HTTP-сервера на порту 8080 для обслуживания статического и динамического содержимого рабочей директории.
twistd dns -p 5553 --hosts-file=hosts
    Запуск DNS-сервера на порту 5553, преобразующего домены из файла с названием hosts, использующего формат файла
    /etc/hosts.
sudo twistd conch -p tcp:2222
    Запуск SSH-сервера на порту 2222. Ключи SSH должны быть установлены отдельно.
twistd mail -E -H localhost -d localhost=emails
    Запуск ESMTP POP3-сервера, принимающего почту для локального узла и сохраняющего ее в директории emails.

```

Утилита `twistd` упрощает создание сервера для тестирования клиентов, при этом используя расширяемый код промышленного уровня.

В этом отношении механизмы развертывания приложений Twisted на основе ТАС-файлов, плагинов и утилиты `twistd` были успешны. Однако, сложилась анекдотичная ситуация, при которой большинство крупномасштабных внедрений фреймворка Twisted заканчивалось необходимостью переработки этих систем управления и мониторинга; их архитектура не соответствовала требованиям системных администраторов. Это обстоятельство было обусловлено фактом отсутствия влияния системных администраторов - людей, которые являются экспертами в области внедрения и сопровождения приложений - на архитектуру в ходе истории развития проекта.

Для проекта Twisted в будущем будет полезно более активное взаимодействие с квалифицированными конечными пользователями при принятии новых архитектурных решений в этой области.

21.3. Взгляд в прошлое и выученные уроки

Проект Twisted недавно отметил свой десятый день рождения. С начала его создания под впечатлением от процесса разработки сетевой игры в начале 2000 годов, он в значительной степени достиг цели, заключающейся создании расширяемого, кроссплатформенного управляемого событиями сетевого фреймворка. Twisted используется в промышленных окружениях компаний от Google и Lucasfilm до Justin.TV и платформы для совместной разработки программного обеспечения Launchpad. Реализации серверов Twisted являются основой множества других приложений с открытым исходным кодом, среди которых BuildBot, BitTorrent и Tahoe-LAFS.

Проект Twisted претерпел несколько кардинальных архитектурных изменений с момента начала его разработки. Одним из важнейших нововведений являлась реализация класса `Deferred`, предназначенного, как было описано выше, для управления ожидаемыми результатами и соответствующими им цепочками функций обратного вызова.

Также было одно важное удаление системы, которое практически не оставило следа в современной реализации: речь идет о системе постоянного хранения данных приложения Twisted.

Система постоянного хранения данных приложения Twisted

Система постоянного хранения данных приложения Twisted (Twisted Application Persistence (ТАР)) предназначалась для сохранения данных конфигурации и состояния приложения в файле, формирование которого происходило с использованием модуля `pickles` из состава Python. Процесс запуска приложения при использовании этой системы состоял из двух стадий:

1. Создание представляющего приложение файла с помощью на данный момент не используемой утилиты `mktar`.
2. Распаковка данных приложения с помощью утилиты `twistd`.

В основу этого процесса были положены принципы, используемые в образах Smalltalk и позволяющие уйти от ограничений специальных языков описания параметров конфигурации, которые не могли использоваться как сценарии, а применение этих принципов было продиктовано желанием описать параметры конфигурации с помощью кода на языке Python.

Использование этих ТАР-файлов сразу же привело к излишним сложностям. Классы фреймворка Twisted могли меняться без изменения экземпляров этих классов, упакованных в файлы. Попытки использования методов или атрибутов классов из новой версии Twisted совместно с упакованными в файл объектами приводили к краху приложений. Было введено понятие "систем обновления", которые должны были обновлять содержимое файлов при изменении версии API, но после этого возникла необходимость поддержки в актуальном состоянии матрицы из систем обновления, версий ТАС-файлов и модульных тестов для учета всех возможностей обновления, хотя всесторонний учет всех изменений интерфейса был так же сложен и приводил к ошибкам.

Файлы ТАР и соответствующие утилиты были лишены поддержки и в конечном итоге удалены из комплекта поставки Twisted и заменены на ТАС-файлы и плагины. Аббревиатура ТАС стала расшифровываться как Twisted Application Plugin (плагин приложения Twisted) и на сегодняшний день о неудавшейся системе хранения данных в Twisted напоминает только несколько фрагментов кода.

Урок, выученный в результате фиаско с системой ТАР заключается в том, что для возможности осуществления разумной поддержки системы хранения данных должна использовать явно установленную схему. В более общем случае это был урок о том, как повышать сложность проекта: при размышлении о создании инновационной системы для решения определенной задачи перед добавлением кода в проект следует удостовериться в том, что сложность рассматриваемого решения понятна и проверена, а также в том, что достоинства системы явно стоят того, чтобы пойти на усложнение проекта.

web2: урок о повторной разработке

Не являясь архитектурным решением, принятое разработчиками проекта решение о повторной реализации подсистемы Twisted Web длительное время оказывало воздействие на имидж проекта Twisted и на возможность улучшения архитектуры других частей кодовой базы сопровождающими проект людьми, поэтому оно заслуживает краткого описания.

В середине 2000 годов разработчики Twisted решили полностью переработать API `twisted.web` в рамках отдельного проекта на основе кодовой базы Twisted с названием `web2`. Проект `web2` должен был содержать множество улучшений по сравнению с `twisted.web`, среди которых полная поддержка протокола HTTP 1.1 и API для работы с потоковыми данными.

Проект `web2` считался экспериментальным, но в итоге начал использоваться основными проектами и даже был выпущен и упакован для дистрибутива Debian. Параллельная разработка `web` и `web2` продолжалась в течение нескольких лет и новые пользователи не понимали того, какой проект стоит использовать из-за существования двух аналогичных проектов и отсутствия понятных рекомендаций на этот счет. Перехода на использование проекта `web2` так никогда и не произошло, а в 2011 году проект `web2` был окончательно удален из кодовой базы проекта и с вебсайта. Некоторые улучшения из проекта `web2` медленно портируются в проект `web`.

Частично по причине событий вокруг проекта `web2`, проект Twisted заработал репутацию сложно структурируемого и не понятного для новичков проекта. Спустя годы сообщество разработчиков Twisted все еще занимается разрушением этого стереотипа.

Выученный с помощью проекта `web2` урок заключается в том, что полная повторная разработка проекта обычно является плохой идеей, но если это происходит, следует удостовериться в том, что сообществу разработчиков понятен долговременный план разработки, а сообщество пользователей имеет один явный выбор реализации для использования во время разработки нового проекта.

В том случае, если в рамках проекта Twisted будет предпринят возврат к прошлому и снова начата разработка `web2`, разработчикам придется внести множество обратно совместимых изменений и

объявить ряд функций устаревшими в рамках `twisted.web` вместо полной повторной разработки проекта.

Соответствие тенденциям развития Интернет

Методы использования нами сети Интернет продолжают развиваться. Решение о реализации множества протоколов в рамках основной кодовой базы Twisted привело к необходимости поддержки кода для этих протоколов. Реализации приходилось развивать в соответствии с изменениями стандартов и введением новых протоколов, при этом придерживаясь строгой политики обратной совместимости.

Проект Twisted в первую очередь развивается силами добровольных разработчиков и ограничивающим разработку фактором является не энтузиазм сообщества, а свободное время разработчиков. Например, спецификация RFC 2616, описывающая протокол HTTP 1.1, была выпущена в 1999 году, а работа по добавлению поддержки протокола HTTP 1.1 в набор реализаций протоколов Twisted началась в 2005 и закончилась в 2009 году. Поддержка протокола IPv6, описанного в спецификации RFC 2460 от 1998 года, находится в стадии реализации, но все еще не добавлена в основную кодовую базу по данным на 2011 год.

Реализации также приходится развивать в ходе изменения интерфейсов поддерживаемых операционных систем. Например, возможность получения уведомлений о событиях с использованием вызова `epoll` была добавлена в Linux 2.5.44 в 2002 году и, для того, чтобы использовать преимущества нового API, в Twisted был добавлен цикл обработки событий на основе `epoll`. В 2007 году компания Apple выпустила версию 10.5 своей операционной системы с именем Leopard, реализация вызова `poll` в которой не поддерживала работу с устройствами, причем этого изменения было достаточно для нарушения работы и блокировки компанией Apple интерфейса `select.poll` в своей сборке интерпретатора Python. Проекту Twisted пришлось предложить [обходное решение](#) этой проблемы и включить его описание в документацию для пользователей.

Иногда темп разработки Twisted оказывается недостаточным для реализации изменений сетевых протоколов и улучшения перемещаются в библиотеки, не относящиеся к основному коду проекта. Например, [проект Wokkel](#), являющийся коллекцией улучшений поддержки протокола Jabber/XMPP фреймворком Twisted, развивался как предназначенный для слияния с проектом Twisted обособленный проект в течение многих лет без перспектив этого слияния. Ввиду того, что в браузеры начали добавлять поддержку нового протокола WebSockets, в 2009 году была предпринята попытка добавления поддержки этого протокола в Twisted, но разработка все же была перенесена в рамки отдельных проектов после решения разработчиков не включать в Twisted код поддержки протокола до того момента, как на основе черновика IETF не будет сформирован стандарт.

Как было сказано, возможность распространения библиотек и дополнений является свидетельством гибкости и расширяемости фреймворка Twisted. Строгая политика разработки с обязательным тестированием, наличие сопроводительной документации и стандартов разработки кода помогают избегать регрессий в рамках проекта и поддерживать обратную совместимость при наличии поддержки большого количества протоколов и платформ. Это зрелый, стабильный проект, который продолжает активно разрабатываться и внедряться.

Twisted может быть и вашим фреймворком для работы с Интернет в течение следующих десяти лет.

22. Фреймворк Yesod

Глава 22 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Yesod является веб-фреймворком, написанным на языке программирования Haskell. В то время как многие популярные веб фреймворки используют динамическую природу своих базовых языков, Yesod использует статические особенности языка Haskell с тем, чтобы создавать безопасный и быстрый код.

Разработка началась около двух лет назад, и с тех пор все еще идет. Фреймворк Yesod в реальных проектах используется еще не в полной мере, не со всеми его изначальными свойствами, появившимся вне фактических, реальных потребностей. Сначала разработка почти полностью напоминала игру одного актера. Примерно через год разработок, когда усилия, делаемые сообществом, переломили ход событий, Yesod превратился в процветающий проект с открытым кодом.

В своей эмбриональной фазе, когда Yesod был невероятно эфемерным и недостаточно определенным, было бы контрпродуктивным попытаться предложить его какой-нибудь команде с тем, чтобы на нем работать. Со временем он стал достаточно стабильным с тем, чтобы быть полезным для других, и это как раз был тот момент, когда следует оценить недостатки некоторых решений, которые были сделаны. Затем мы внесли значительны изменения в интерфейс API, обращенный к пользователю, с тем, чтобы сделать его более полезным, и все это зафиксировали в версии 1.0.

Вы можете спросить: почему еще один веб-фреймворк? Давайте вместо этого ответим на другой вопрос: Зачем использовать язык Haskell? Похоже, что большая часть мира вполне счастлива с одним из языков, относящихся к следующим двум стилям:

- Статически типизированные языки, такие как Java, C# и C++. В этих языках обеспечивается хорошая скорость работы и безопасность типов, но при программировании они более громоздки.
- Динамически типизированные языки, такие как Ruby и Python. Эти языки значительно увеличивают производительность (по крайней мере в краткосрочной перспективе), но работают медленно и очень слабо поддерживаются компиляторами с точки зрения проверки правильности. Решением этого последнего аспекта занимается юнит-тестирование. Мы вернемся к нему позже.

Такое деление ложное. Нет никаких причин, почему статически типизированные языки должны быть таким неуклюжими. Haskell способен быть столь же выразительным, как Ruby и Python, оставаясь при этом строго типизированным языком. На самом деле, система типов в Haskell позволяет обнаружить гораздо больше ошибок, чем язык Java и ему подобные. Полностью устраниены исключения, возникающие в случаях, когда значение указателя равно null; неизменяемые структуры данных упрощают обдумывание кода и упрощают параллельное программирование.

Так почему же язык Haskell? Это эффективный, дружественный для разработчика язык, в котором много проверок времени компиляции, что гарантирует корректность программ.

Целью проекта Yesod является распространение сильных сторон языка Haskell в направлении веб-разработки. Yesod стремится сделать код как можно более кратким. Во время компиляции на корректность проверяется, насколько это возможно, каждая строчка кода. Компилятор делает все это за вас вместо того, чтобы требовать больших библиотек юнит-тестов для проверки основных свойств. В глубине Yesod используются многие современные технологии повышения производительности, которые заставят летать ваш высокоуровневый код.

22.1. Сравнение с другими фреймворками

В общих чертах, Yesod больше похож на ведущие фреймворки, такие как Rails и Django., а не отличается от них. Он, в целом, соответствует парадигме Model-View-Controller (MVC), имеет систему шаблонов, в которой внешнее представление отделено от логики, предложена система объ-

ектно-реляционного отношения (Object-Relational Mapping — ORM) и есть контролер, предназначенный для реализации навигации.

Дьявол кроется в деталях. В Yesod делается попытка выявить основное количество ошибок на фазе компиляции, а не во время выполнения, а также для автоматически отлавливать как ошибки, так и изъяны в безопасности через систему типов. Хотя Yesod пытается оставаться дружественным высокоуровневым API, в нем для достижения высокой производительности используется ряд новых методов из мира функционального программирования, и эти внутренние особенности не скрыты от разработчиков.

Основной архитектурной проблемой в Yesod является балансирование этих двух, казалось бы противоречащих друг другу целей. Например, нет ничего революционного подхода в методе маршрутизации (используются типобезопасные адреса URL), применяемом в Yesod. Исторически, внедрение такого решения было утомительным процессом, подверженным ошибкам. Инновация, имеющаяся Yesod, состоит в использовании шаблона Template Haskell(вариант генерации кода) для автоматизации типовых операций, необходимых для начальной самозагрузки процесса. Аналогичным образом, в течение долгого времени были повсюду типобезопасные страницы HTML; Yesod пытается выделить те аспекты, которые дружественны для разработчика и присутствуют в обычных языках работы с шаблонами, и, при этом, сохранить всю мощь, обеспечиваемую безопасностью типов.

22.2. Интерфейс веб-приложений

Веб-приложениям необходим некоторый способ общения с сервером. Одним из возможных способов является встраивание сервера непосредственно во фреймворк, но такой подход обязательно будет ограничивать ваши возможности по развертыванию веб-приложения и ведет к созданию плохих интерфейсов. Для решения этой проблемы во многих языках создавались стандартные интерфейсы: в языке Python есть WSGI, а в Ruby есть Rack. В Haskell, у нас есть WAI: Web Application Interface - веб-интерфейс приложений.

Интерфейс WAI не предназначен для использования в качестве интерфейса высокого уровня. У него есть две конкретные цели: обеспечить единообразность и производительность. Оставаясь единообразным, интерфейс WAI поддерживает работу с различными фоновыми процессами начиная от автономно работающих серверов и до технологии CGI в старом стиле, и даже до непосредственного использования Webkit при создании приложений, работающих на рабочем столе. Что касается производительности, то мы представим вам ряд интересных функций языка Haskell.

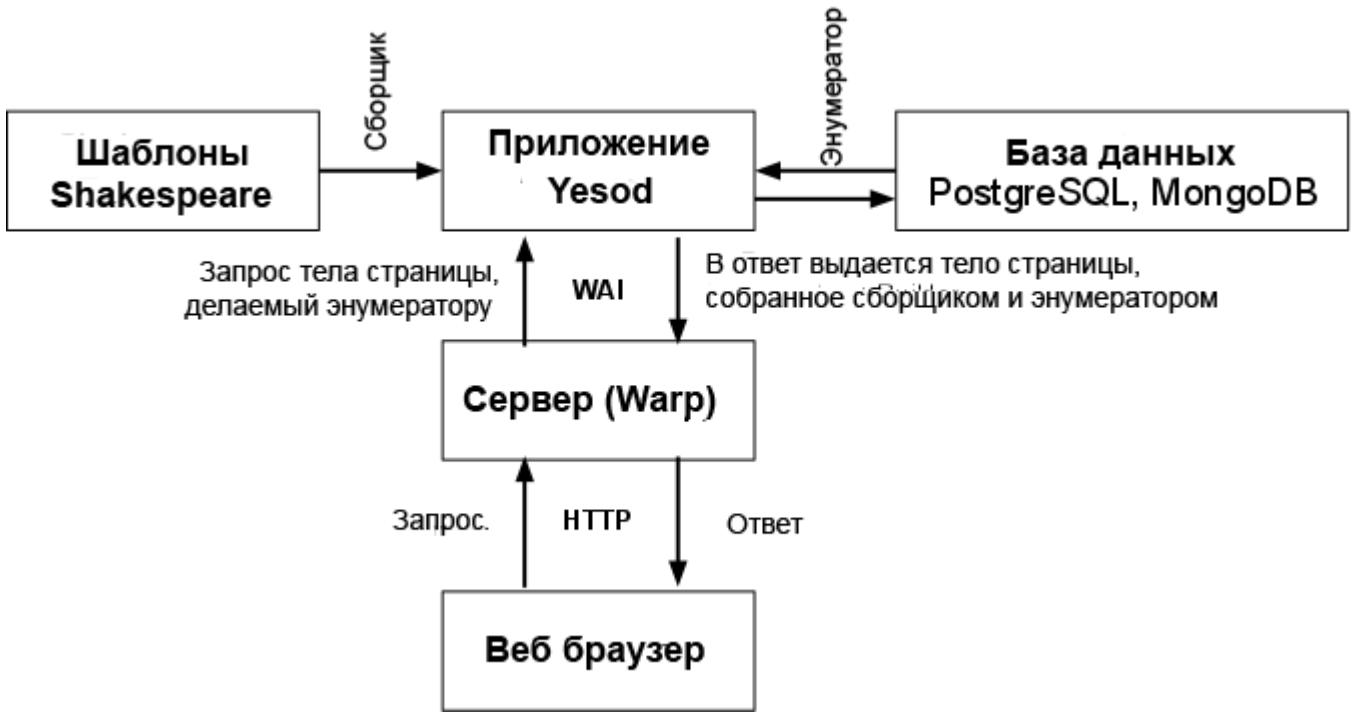


Рис.22.1: Общая структура приложения Yesod

Типы данных

Одно из самых больших преимуществ языка Haskell и то, что мы пытаемся максимально использовать в Yesod, это строгая статическая типизация. Перед тем, как начать писать код для того, чтобы что-нибудь решить, нам нужно будет подумать о том, как будут выглядеть данные. Интерфейс WAI является прекрасным примером этой парадигмы. Основная концепция, которую мы хотим выразить, является приложение. Самым основным базовым выражением является функция, к которой делается запрос и от которой возвращается ответ. На языке Haskell:

```
type Application = Request -> Response
```

Только возникает вопрос: как должен выглядеть запрос `Request` и ответ `Response`? В запросе есть ряд информационных фрагментов, но самым основным являются запрашиваемый путь, строка запроса, заголовки запроса и тело запроса. А в ответе должны быть только три компонента: код состояния, заголовки ответа и тело ответа.

Как мы представляем себе что-то, похожее на строку запроса? В языке Haskell соблюдается строгое разделение между двоичными и текстовыми данными. Первые можно представить с помощью `ByteString`, а вторые — с помощью `Text`. Оба типа являются хорошо оптимизированными типами данных, для которых предоставляется высокоуровневый безопасный интерфейс API. В случае строки запроса мы сохраняем необработанные байт данные, передаваемые по сети в виде типа `ByteString`, а проанализированные декодированные значения — в виде типа `Text`.

Потоки

Тип `ByteString` представляет собой отдельно взятый буфер памяти. Если бы мы по наивности использовали простой тип `ByteString` для передачи тела запроса и получения тела ответа, то наши приложения никогда нельзя было бы масштабировать для работы с большими запросами и ответами. Вместо этого мы используем технологию, называемую `enumerator` (перечисление), очень похожую по концепции на `generators` (генераторы) в языке Python. Наше приложение `Application` становится потребителем потока объектов типа `ByteStrings`, представляющих собой тело входящего запроса, и создает отдельный поток для ответа.

Теперь нам нужно немного пересмотреть наше определение приложения Application. Приложение Application выдает значение типа Request, в котором содержатся заголовки, строка запроса и т.д., и будет получать поток объектов типа ByteString, из которых состоит ответ Response. Таким образом, пересмотренное определение приложения будет Application следующим:

```
type Application = Request -> Iteratee ByteString IO Response
```

Обозначение IO просто указывает какие типы побочных эффектов приложение может выполнять. В случае IO, оно может выполнять любые виды взаимодействий с внешним миром, что является очевидной необходимостью для подавляющего большинства веб-приложений.

Сборщик

Весь фокус нашего арсенала состоит в том, как мы создаем буферы с нашими ответами. У нас есть здесь два конкурирующих пожелания: минимизация количества системных вызовов и минимизация количества копирований буфера. С одной стороны, мы хотим свести к минимуму количество системных вызовов при передаче данных через сокет. Для этого нам нужно хранить исходящие данные в буфере. Тем не менее, если мы сделаем этот буфер слишком большим, то мы исчерпаем нашу память и замедлим время отклика приложения. С другой стороны, мы хотим свести к минимуму количество копирований, когда данные копируются между буферами, предпочтительно копировать только один раз из буфера источника данных в буфер назначения.

В языке Haskell есть решение, которое называется *сборщиком (builder)*. Сборщик представляет собой инструкцию о том, как заполнять буфер памяти, например: разместить пять байтов «hello» в следующей свободной позиции. Вместо передачи потока буферов памяти на сервер, приложение WAI обрабатывает поток этих инструкций. Сервер берет поток и использует его так, чтобы оптимальным образом использовать буфера памяти, имеющие определенный размер. Как только буфер будет заполнен, сервер осуществляет системный вызов с тем, чтобы послать данные по сети, а затем начинает заполнение буфера следующего.

Оптимальный размер буфера будет зависеть от многих факторов, например, от размера кэш-памяти. Библиотека blaze-builder, на основе которой все это выполняется, прошла через существенное тестирование производительности прежде, чем был найден наилучший компромисс.

В теории, такая оптимизация может быть выполнена в самом приложении. Однако, когда этот подход кодируется в интерфейсе, мы можем просто добавлять заголовки ответа в тело ответа. Результатом является то, что для ответов малых и средних размеров, весь ответ может быть отправлен с помощью одного системного вызова и только с одним копированием в память.

Обработчики

Теперь, когда у нас есть приложение, мы должны каким-то образом его запустить. С точки зрения интерфейса WAI, это делает *обработчик (handler)*. В WAI есть некоторые базовые стандартные обработчики, например, автономные серверы Warp (о нем будет рассказано ниже), FastCGI, SCGI и CGI. Такой спектр серверов позволяет запускать приложения WAI приложений, которые будут работать начиная от специально выделенных серверов и до виртуального хостинга. Но, в дополнение к этому, в WAI есть еще несколько интересных возможностей:

- **Webkit:** Этот движок встроен в сервер Warp и вызывается при обращении к QtWebKit. Когда запускается сервер, то затем открывается новое отдельное окно браузера и у нас есть приложение, работающее на рабочем столе.
- **Launch:** Это несколько упрощенный вариант Webkit. Необходимость развертывать библиотеки Qt и Webkit может быть несколько обременительной, так что мы вместо этого просто запускаем браузер, используемый пользователем по умолчанию.

- **Test:** Даже тестирование рассматривается как обработчик. В конце концов, тестирование просто акт запуска приложения и проверки его ответов.

Большинство разработчиков, скорее всего, будут использовать Warp. Это достаточно легковесный сервер, которого будет достаточно для тестирования. Он не требует конфигурационных файлов, нет иерархии папок и долгоиграющего процесса, принадлежащего администратору. Это простая библиотека, которая компилируется в вашем приложении или запускается с помощью интерпретатора Haskell. Warp является невероятно быстрым сервером с защитой от всех видов атак, например, Slowloris или бесконечного списка заголовков. Warp может быть единственным веб-сервером, который вам нужен, хотя все будет также в порядке, если его поместить за прокси-сервером HTTP.

С помощью бенчмарка PONG было измерено количество запросов, состоящих из четырехбайтового тела «PONG» и обрабатываемого за секунду на различных серверах. На графике, показанном на рисунке 22.2, Yesod измеряется как фреймворк, работающий поверх Warp. Как видно, сервера Haskell (Warp, Happstack и Snap) лидируют.

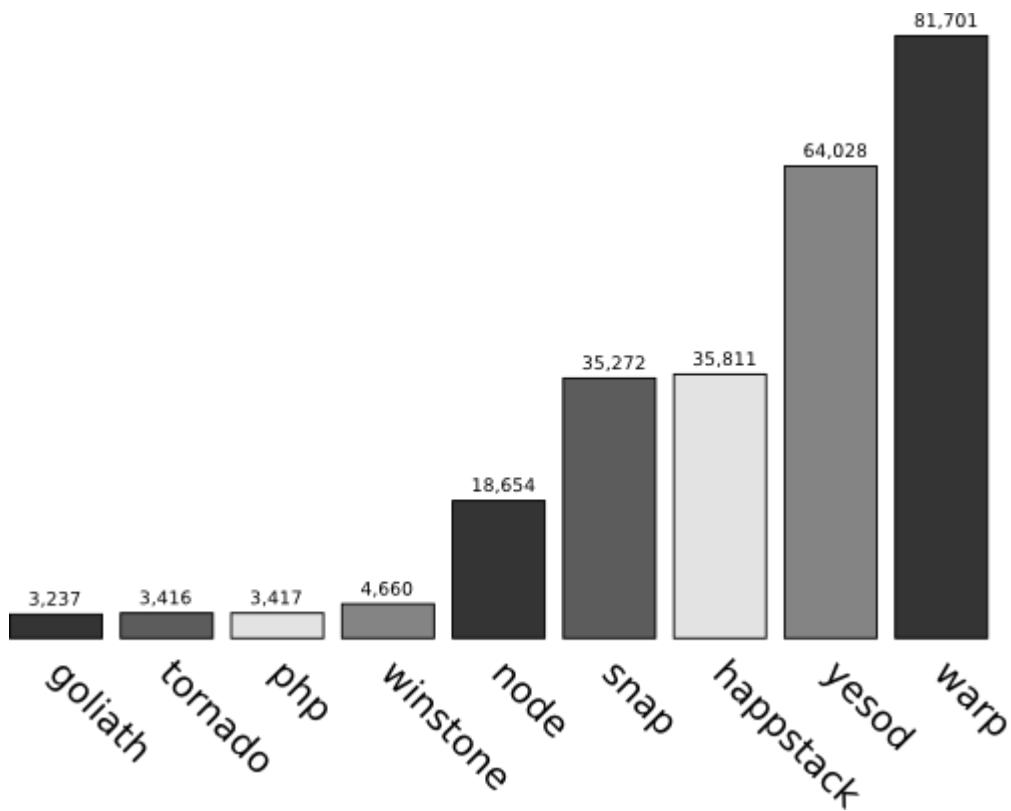


Рис.22.2: Бенчмарк PONG для сервера Warp

Большинство причин такой скорости сервера Warp уже были изложены в общей характеристике интерфейса WAI: счетчики, сборщики и упакованные типы данных. Последняя часть головоломки заключается в многопоточности времени выполнения компилятора Glasgow Haskell Compiler (GHC's). GHC, флагманский компилятор языка Haskell, использует легковесные потоки с незначительным потреблением ресурсов. В отличие от системных потоков, можно запустить тысячи таких потоков без серьезной загрузки по производительности. Таким образом, в каждом соединении Warp обрабатывается свой собственный поток со слабым потреблением ресурсов.

Следующая хитрость состоит в асинхронном вводе / выводе. В любом веб-сервере, в котором нужно масштабирование до десятков тысяч запросов в секунду, требуется некоторый тип асинхронной связи. В большинстве языков, это связано с участием сложного в программировании механизма обратного вызова. GHC позволяет нам обмануть систему: мы программируем как будто мы используем синхронный интерфейс API, а GHC автоматически переключается между различными потоками, ожидающими своей очереди.

Глубже GHC пользуется всем, что предоставляет хостовая операционная система, например, `kqueue`, `epoll` и `select`. В результате мы получаем производительность системы ввода / вывода, использующей события, и не беспокоимся о кроссплатформенных вопросах и не пишем в стиле, ориентированном на обратные вызовы.

Промежуточный слой middleware

Между обработчиками и приложениями у нас есть промежуточный слой `middleware`. Технически он представляет собой преобразователь приложений *application transformer*: он берет одно приложение и возвращает новое. Это определяется следующим образом:

```
type Middleware = Application -> Application
```

Лучший способ понять назначение промежуточного слоя - это рассмотреть некоторые распространенные примеры:

- `gzip` автоматически сжимает ответ, получаемый от приложения.
- `jsonp` автоматически преобразует ответы JSON в ответы JSON-P, когда клиент представляет параметр функции обратного вызова.
- `autohead` будет генерировать соответствующие ответы HEAD на основе ответа GET, поступившего от приложения.
- `debug` будет на каждый запрос выдавать отладочную информацию в консоль или в журнал.

Идея заключается в том, чтобы вынести из приложений за скобки общий код и позволить его использовать совместно нескольким приложениям. Обратите внимание, что, исходя из определения такого промежуточного слоя, мы можем легко расширять такие возможности. Общая схема работы промежуточного слоя состоит в следующем:

1. Берем значение запроса и применяем некоторые модификации.
2. Передаем модифицированный запрос в приложение и принимаем ответ.
3. Модифицируем ответ и возвращаем его обработчику.

В случае, если промежуточных слоев будет несколько, то вместо того, чтобы передавать данные в приложение или в обработчик, можно будет передавать их в промежуточный слой, лежащий глубже, или во внешний слой, лежащий, соответственно, выше.

Тесты wai-test

Никакая статическая типизация не позволит устраниć необходимость тестирования. Мы все знаем, что автоматизированное тестирование является необходимостью для любого серьезного приложения. `wai-test` является рекомендуемым подходом к тестированию приложений с интерфейсом WAI. Поскольку запросы и ответы являются простыми типами данных, легко сформировать искусственный запрос, передать его в приложение и проверить, какой будет ответ. В `wai-test` просто предлагаются некоторые удобные функции для тестирования общих свойств, таких как наличие заголовка или кода состояния.

22.3. Шаблоны

В типичной парадигме Model-View-Controller (MVC), одной из целей является отделение логики обработки данных от представления данных. Часть этого разделения достигается за счет использования шаблонов языка. Однако, есть много разных способов решения этого вопроса. На одном конце спектра, например, в PHP / ASP / JSP вам разрешается вставлять в ваш шаблон произвольный код. На другом конце, у вас есть системы, такие как `StringTemplate` и `QuickSilver`, в которые

передаются некоторые аргументы и нет никакого другого способа взаимодействия с остальной частью программы.

В каждой системе есть свои плюсы и минусы. Когда есть более мощная система шаблонов, то это может быть очень удобно. Нужно показать содержимое таблицы базы данных? Нет проблем, вытащите ее с помощью шаблона. Тем не менее, такой подход может быстро привести к запутанному коду, в котором обновления курсора базы данных перемежается с генерацией кода языка HTML. Это можно часто видеть в плохо написанных проектах ASP.

Хотя слабые системы шаблонов упрощают кодирование, они также заставляют делать очень много лишней работы. Часто вам будет нужно не только сохранить ваши исходные значения в типах данных, но и создавать словари значений, которые будут передаваться в шаблон. Поддержка такого кода является непростой задачей и, как правило, у компилятора нет никакого способа помочь вам.

Семейство Yesod языков шаблонов, Shakespearean (Шекспировских) языков, стремится к золотой середине. Используя стандартную в языке Haskell прозрачность ссылок, мы можем быть уверены, что наши шаблоны не дают побочных эффектов. Тем не менее, у них у всех все еще есть полный доступ ко всем переменным и функциям, доступным в коде Haskell. Кроме того, поскольку они во время компиляции полностью проверяются как на корректность и диапазон значений, так и на безопасность типов, вероятность наличия ошибок из-за опечаток гораздо менее вероятна, чем когда вы просто просматриваете код с целью выявить ошибки.

Почему Shakespeare (Шекспир)?

Язык HTML, Hamlet (Гамлет), был первым написанным языком, синтаксис которого базировался на языке Haml. Поскольку это было во времена «сокращенного варианта» Haml, язык Hamlet казался более подходящим. Когда мы добавили CSS и Javascript, мы решили продолжить тему именования и назвали варианты Cassius (Кассий) и Julius (Юлий). На данный момент, язык Hamlet не похож Haml, но, тем не менее, название прижилось.

Типы

Одной из важнейших тем в Yesod является правильное использование типов, что позволяет сделать жизнь разработчиков проще. В шаблонах Yesod у нас есть два основных примера:

1. Все содержимое, внедренное в шаблон Hamlet (Гамлета), должно иметь тип `Html`. Как мы увидим позже, это заставляет нас, когда это необходимо, избегать использовать опасный код HTML, например, случайного двойного преобразования.
2. Вместо того, чтобы вставлять адрес URL непосредственно в наш шаблон, у нас есть типы данных, известные как типобезопасные адреса, которые представляют собой маршруты в нашем приложении.

В качестве реального примера, предположим, что пользователь отправляет в приложение с помощью формы свое имя. Эти данные будут представлены типом данных `Text`. Теперь мы хотели бы отобразить на странице эту переменную, которая называется `name`. Система типов во время компиляции предотвращает просто вставить ее в шаблон Hamlet, поскольку она не относится к типу `Html`. Вместо этого мы должны ее как-то преобразовать. Для этого есть две функции преобразования:

1. Преобразование `toHtml`, которое автоматически выбрасывает любые лишние вставки. Таким образом, если пользователь отправляет строку `<script src="http://example.com/evil.js"></script>`, символы «меньше чем» будут автоматически преобразованы в `<`

2. Преобразование `preEscapedText`, с другой стороны, оставит содержимое в том виде, как оно есть.

Таким образом, в случае недопустимого ввода данных, сделанного, возможно, зловредным пользователем, в качестве рекомендуемого нами подхода будет использование преобразования `toHtml`. С другой стороны, скажем, у нас есть несколько статических документов HTML, хранящихся на нашем сервере, которые мы хотели бы вставлять в некоторые страницы без всяких изменений. В этом случае, мы могли бы загрузить их в значение `Text`, а затем применить преобразование `preEscapedText`, что позволит нам избежать всяких преобразований.

По умолчанию, Hamlet будет использовать функцию `toHtml` для любого контента, который вы пытаетесь вставить. Таким образом, вам нужно явно выполнить преобразование только в случае, если преобразование вам явно не нужно. Это следует изречению о том, что ошибиться лучше в сторону излишней осторожности.

```
name <- runInputPost $ ireq textField "name"
snippet <- readFile "mysnippet.html"
return [hamlet|
    <p>Welcome #{name}, you are on my site!
    <div .copyright>#{preEscapedText snippet}
|]
```

Первым шагом в типобезопасных адресах URL является создание тип данных, который представляет все маршруты на вашем сайте. Скажем, у вас есть сайт для отображения чисел Фибоначчи. Сайт будет иметь отдельную страницу для каждого числа в последовательности и, плюс, домашнюю страницу. Это может быть сделано с помощью типа данных в языке Haskell следующим образом:

```
data FibRoute = Home | Fib Int
```

Затем мы могли бы создать страницу, например, следующим образом:

```
<p>You are currently viewing number #{show index} in the sequence. Its value is #{fib index}.
<p>
    <a href=@{Fib (index + 1)}>Next number
<p>
    <a href=@{Home}>Homepage
```

Тогда все, что нам нужно, это некоторая функция, которая преобразует типобезопасный URL в строковое представление. В нашем случае, это может выглядеть примерно так:

```
render :: FibRoute -> Text
render Home = "/home"
render (Fib i) = "/fib/" ++ show i
```

К счастью для разработчика, все шаблоны, определяющие и отображающие типы данных для типобезопасных адресов URL, обрабатывается в Yesod автоматически. Позже мы рассмотрим это более подробно.

Другие языки

В дополнение к языку Hamlet предлагаются еще три других языка: Julius (Юлий), Cassius (Кассий) и Lucius (Луций). Julius используется для Javascript, однако, это простой сквозной язык, который просто позволяет делать вставки. Другими словами, за исключением случайного использования синтаксиса вставок, любая часть Javascript может быть убрана из Julius и синтаксис останется действительным.

ствительным. Например, для тестирования производительности Julius, с помощью него был добавлен фреймворк Jquery и при этом не возникло никаких проблем.

Два других языка являются альтернативой синтаксису CSS. Те, кто знаком с разницей между Sass и Less увидят это различие немедленно: в Cassius в качестве разделителей используются пробелы, в то время как в Lucius использует фигурные скобки. Lucius, на самом деле, является расширением CSS, то есть все допустимые файлы CSS также будут допустимыми в Lucius. В дополнение к возможности вставки текста, есть некоторые вспомогательные типы данных, предлагаемые для моделирования размеров и цвета. Также в этих языках работают типобезопасные адреса URL, что удобно при определении фоновых изображений.

Помимо безопасности типов и проверок во время компиляции, о чем упоминалось выше, наличие специализированных языков для CSS и Javascript предоставляет нам несколько других преимуществ:

- При разработке приложений все CSS и Javascript компилируются в окончательный исполняемый файл, что повышает производительность (благодаря отсутствию ввода/вывода) и упрощает развертывание приложения.
- Благодаря тому, что все это базируется на эффективном конструкторе сборщика, который был описан ранее, прорисовка шаблонов осуществляется очень быстро.
- Есть встроенная поддержка автоматического включения CSS и Javascript в окончательные веб-страницы. Мы столкнемся с этим более подробно, когда далее будем рассматривать виджеты.

22.4. Хранение данных с помощью Persistent

Большинство веб-приложений будут сохранять информацию в базе данных. Традиционно, это означает некоторый вариант SQL - базы данных. В этой связи, в Yesod продолжается давняя традиция с PostgreSQL в качестве нашего наиболее часто используемого хранилища данных. Но, как мы видели в последнее время, вопрос долговременного хранения данных не всегда решается с помощью SQL-базы. Поэтому проект Yesod был разработан таким образом, чтобы он мог также хорошо работать с базами данных вида NoSQL, и поставляется с MongoDB в качестве отличного решения в качестве хранилища данных.

Результатом этого дизайнерского решения является Persistent, компонент хранения данных, используемый в Yesod. Есть, в действительности, два стиля использования Persistent: сделать его настолько нейтральным к хранимым типам, насколько это возможно, и предложить пользователю полную проверку типов.

В то же самое время, мы в полной мере осознаем, что невозможно полностью оградить пользователей от всех деталей, связанных с хранением данных. Поэтому мы предлагаем два варианта обходных путей:

- Функции, связанные с особенностями хранилища данных. Например, в Persistent предлагаются возможность работы с объединениями SQL и списками и хэш-таблицами MongoDB. Предупреждается, что ее использование нарушит переносимость приложения, но если вам нужны эти функции, то они есть.
- Простой доступ, позволяющий напрямую выполнять запросы к хранилищу данных. Мы не считаем, что нужно с помощью некоторой абстракции реализовывать все функциональные возможности, предоставляемые базовой библиотекой. Если вам просто нужно написать таблицу, связанную с подзапросом в SQL, то - вперед.

Терминология

Самым примитивным типом данных в компоненте Persistent является PersistValue. С его помощью представлены все данные, непосредственно хранящиеся в базе данных, например, числа, даты или строки. Конечно, иногда у вас будут некоторые более дружественные типы данных, которые вы захотите хранить, например, HTML. Для этого у нас есть класс PersistField. Внутри базы данных PersistField выражается через PersistValue.

Все это очень хорошо, но мы хотим объединять различные поля вместе в одну большую картину. Для этого у нас есть класс PersistEntity, который является, по существу, коллекцией классов PersistField. И, наконец, у нас есть класс PersistBackend, в котором описывается, как создавать, читать, обновлять и удалять все эти сущности.

В качестве практического примера рассмотрим хранение в базе данных информации о некоторой персоне. Мы хотим хранить имя, день рождения и изображение профиля (файл PNG). Мы создаем новую сущность Person с тремя полями: Text, Day и PNG. Каждый из них хранится в базе данных с использованием другого конструктора PersistValue: PersistText, PersistDay и PersistByteString, соответственно.

Что касается первых двух отображений, то в них нет ничего удивительного, но последнее из них является интересным. Нет конкретного конструктора для хранения содержимого PNG в базе данных, поэтому вместо этого мы пользуемся более универсальным типом (ByteString, который всего лишь является последовательностью байтов). Мы могли бы использовать тот же самый механизм для хранения других типов произвольных данных.

Распространен лучший практический способ хранения изображений, когда данные хранятся в файловой системе, а в базе данных хранится только путь к изображению. Мы не выступаем против использования такого подхода, а используем хранение изображений в базе данных скорее в качестве иллюстративного примера.

Как все это представлено в базе данных? В качестве примера рассмотрим SQL: сущность Person становится таблицей с тремя колонками (имя, дата рождения и изображение). Каждое поле хранится в виде различных типов SQL: тип Text становится типом VARCHAR, тип Day становится типом Date, а тип PNG становится типом BLOB (или BYTEA).

История для MongoDB очень похожа. Тип Person становится его собственным документом *document*, а его три поля каждой становится полем *field* в MongoDB. В MongoDB не нужны типы данных и не нужно создавать схему.

Persistent	SQL	MongoDB
PersistEntity	Таблица	Документ
PersistField	Столбец	Поле
PersistValue	Тип столбца	Нет

Безопасность типов

Компонент Persistent обрабатывает все данные, решая все проблемы поиска и доступа так, что они для вас будут незаметны. Как пользователь Persistent, вы получаете возможность полностью игнорировать тот факт, что тип Text становится типом VARCHAR. Вы можете просто объявить типы данных и использовать их.

Каждое взаимодействие с Persistent является строго типизированным. Это позволяет предотвратить случайное запоминание номера в поле даты; компилятор это не допустит. В такой ситуации просто исчезают целые классы тонких ошибок.

Нигде сила строгой типизации не проявляется сильнее, чем при рефакторинге. Скажем, у вас в базе данных хранились значения возраста пользователей, и вы понимаете, что в действительности вам вместо них нужно хранить дни рождения. Вы можете сделать одно изменение строки в файле декларации персон, запустить компиляцию и автоматически найти каждую строчку кода, которую следует обновить.

В большинстве динамически типизированных языками, а также в их веб фреймворках, рекомендуется подход к решению этого вопроса, состоящий в написании юнит-тестов. Если тесты покрывают весь код, то, запустив тесты, вам сразу же станет видно, какой код должен быть обновлен. Это все хорошо и правильно, но это более слабое решение, чем использование настоящих типов данных:

- Все это основывается на том, что тест покрывает весь код. Для этого требуется дополнительное время и, что еще хуже, для этого требуется код шаблона, который компилятор должен суметь сделать для вас.
- Вы можете быть идеальным разработчиком, который никогда не забывает написать тест, но можете ли вы сказать то же самое о каждом, кто имеет отношение к вашему коду?
- Даже 100% тестовое покрытие не гарантирует, что вы действительно проверили каждый случай. Доказано, что когда вы все это делаете, вы протестируете каждую строку кода.

Межбазовый синтаксис

Создать схему SQL, которая работает для нескольких движков SQL, может оказаться достаточно сложным. Как создать схему, которая также будет работать с базой данных non-SQL, например, с MongoDB?

Компонент позволяет определять ваши сущности в синтаксисе высокого уровня и автоматически создавать для вас схему SQL. В случае MongoDB, мы в настоящее время используем подход без использования схемы. Также Persistent обеспечивает, чтобы ваши типы данных в языке Haskell точно соответствовали определениям в базе данных.

Кроме того, когда есть вся эта информация, Persistent может для вас автоматически выполнять более сложные функции, такие как миграция данных.

Миграция

В Persistent не только по мере необходимости создаются файлы схем данных, но и автоматически выполняется, если это возможно, миграция базы данных. Модификация баз данных является одним из менее развитых частей стандарта SQL, и поэтому в каждом движке этот процесс происходит по-разному. В результате в каждом компоненте Persistent определяется свой собственный набор правил миграции данных. В PostgreSQL, в котором есть богатый набор правил ALTER TABLE, мы этим пользуемся достаточно широко. Поскольку в SQLite таких функциональных возможностей недостаточно, мы решили создавать временные таблицы и выполнять копирование строк. Подход в MongoDB, в котором отсутствует схема данных, означает, что поддержка миграции не требуется.

Эта возможность специально ограничена с тем, чтобы предотвратить любые потери данных. Нельзя удалять какие-либо столбцы автоматически, вместо этого вы получите сообщение об ошибке, указывающее вам на небезопасные операции, которые необходимы для продолжения действия. Затем вам будет предоставлена возможность либо вручную запустить операцию SQL, которая вам будет предоставлена, либо изменить модель данных с тем, чтобы избежать опасного поведения.

Отношения

Компонент Persistent, по своей природе, не является реляционным, что означает, что в движке не поддержка поддержка реализации отношений. Тем не менее, во многих практических ситуациях нам может потребоваться использовать отношения. В этих случаях разработчикам будет предоставлен к ним полный доступ.

Предположим, что нам теперь с каждым пользователем необходимо хранить список его навыков. Если бы мы писали приложение, специально предназначенное для MongoDB, мы могли бы идти дальше и просто хранить этот список как новое поле в исходной сущности Person. Однако такой подход не будет работать в SQL. В SQL, мы называем такие отношения отношениями типа «один-ко-многим».

Идея состоит в том, чтобы создать ссылку на «одну» сущность (персону) в «множественной» сущности (навыках). Тогда, если мы хотим найти все навыки, которые есть у человека, мы просто находим все навыки, в которых есть ссылка на эту персону. В соответствие с этой ссылкой мы получаем идентификатор ID. Заметим, что, как вы могли бы ожидать, эти идентификаторы являются типобезопасными. Типом данных для идентификатора Person ID является тип PersonId. Таким образом, чтобы добавить наш новый навык, нам нужно в нашем определении сущности просто добавить следующее:

```
Skill
  person PersonId
  name Text
  description Text
  UniqueSkill person name
```

Эта концепция типа данных ID используется везде в Persistent и в Yesod. Вы можете на основе ID организовать диспетчеризацию объектов. В таком случае, Yesod автоматически будет искать и преобразовывать маршрут из текстового представления ID в его внутреннее представлению, по ходу дела выявляя синтаксические ошибки. Эти идентификаторы ID используются для выполнения операций поиска и удаления с использованием функций `get` и `delete`, а также для получения результата операций вставки и выборки значений с использованием функций `insert` и `selectList`.

22.5. Yesod

Если мы посмотрим на типичную парадигму «Модель-Представление-Контроллер» («Model-View-Controller» - MVC), то Persistent является моделью, а Shakespeare является представлением. Тогда все, что остается Yesod, это выступать в роли контроллера.

Самая основная особенность Yesod это - маршрутизация. Она позволяет иметь декларативный синтаксис и типобезопасную диспетчеризацию обращений. На основе этого в Yesod создано много других возможностей: генерация потокового контента, виджеты, интернационализация, статические файлы, формы и аутентификация. Но основной особенностью, которая добавлена в Yesod, в действительности является маршрутизация.

Это многоуровневый подход облегчает пользователям обмениваться различными компонентами системы. Некоторых не интересует использование Persistent. Для них в ядре системы нет ничего, даже упоминающего о Persistent. Точно также, хотя аутентификация и сохранение файлов со статистикой используются часто, эти функции нужны не каждому.

С другой стороны, многие пользователи *хотят* пользоваться всеми этими функциями. И делают это, включая все оптимизации, имеющиеся в Yesod, что не всегда просто. Чтобы упростить процесс, в Yesod также предлагается специальный инструментальный набор, с помощью которого настраивается базовая часть сайта с наиболее часто используемыми возможностями.

Маршруты

Учитывая то, что маршрутизация на самом деле является основной функцией Yesod, давайте начнем с нее. Синтаксис маршрутизации очень прост: *шаблон ресурса*, имя и методы запросов. Например, простой блог-сайт может выглядеть следующим образом:

```
/ HomepageR GET
/add-entry AddEntryR GET POST
/entry/#EntryId EntryR GET
```

В первой строке определена домашняя страница. Здесь говорится - «Я представляю собой корневой каталог домена, мое имя HomepageR и я отвечаю на запросы GET». Завершающий символ «R» в именах ресурсов является просто общепринятым соглашением, в нем не закладывается никакого общего смысла, кроме как просигнализировать разработчику о том, что нечто является маршрутом.

Во второй строке определена страница добавочной записи. На этот раз мы отвечаем на оба запроса GET и POST. Вы удивитесь, почему в Yesod, в отличие от большинства фреймворков, требуется, чтобы вы явно указывали методы ваших запросов. Причина в том, что Yesod старается придерживаться принципов RESTful настолько, насколько это возможно, а запросы GET и POST по смыслу действительно очень различные. Мало того, что вы устанавливаете эти два метода по отдельности, затем вы отдельно определяете для них функции обработчиков. В действительности, это дополнительная функция в Yesod. Если вы хотите, вы можете убрать из списка названия методов и функции ваших обработчиков будут использоваться для всех методов запроса.

Третья строка немного интереснее. После второго «слеша» у нас есть #EntryId. Таким образом определяется параметр EntryId. Мы уже ссылались на эту функцию в разделе о компоненте Persistent: Yesod будет автоматически строить компонент, представляющий собой путь к соответствующему значению ID. Предположим, что на серверной стороне у нас используется SQL (о Mongo поговорим позже), и если пользователь сделает запрос /entry/5, то функции обработчика будет вызвана с аргументом EntryId 5. Но если пользователь сделает запрос /entry/some-blog-post, то Yesod вернет значение 404.

Очевидно, что это также возможно в большинстве других веб фреймворков. Например, в подходе, применяемом в Django, можно использовать регулярные выражения для проверки соответствия маршрутов, например, `r"/entry/(\d+)".` Однако подход, применяемый в Yesod, имеет ряд преимуществ:

- Ввод «EntryId» семантически гораздо более удобен/дружественен для разработчика, чем регулярное выражение.
- С помощью регулярных выражений нельзя выразить все (или по крайней мере, нельзя это сделать лаконично). В Yesod мы можем использовать /calendar/#Day; вы хотите набрать регулярное выражение для того, чтобы сравнивать даты в ваших маршрутах?
- Yesod также автоматически находит для нас пути. В случае использования календаря, наша функция обработки получит значение Day. В эквиваленте в Django, функция получит кусок текста, по которому нужно будет выполнять поиск самостоятельно. Это утомительно, это требуется каждый раз повторять и это неэффективно.
- До сих пор мы предполагали, что идентификатор базы данных ID является простой строкой цифр. Но что, если он более сложен? В MongoDB, например, используются идентификаторы GUID. В Yesod ваш запрос #EntryId все равно будет работать, а система типов проинструктирует Yesod, как анализировать маршрут. В системе регулярных выражений, вам придется пройти по всем вашим маршрутам и заменить \d+ на чудовищно сложное регулярное выражение, необходимое для сравнения идентификаторов GUID.

Адреса типобезопасных URL

Такой подход к маршрутизации порождает одну из самых мощных функций Yesod: типобезопасные адреса URL. Вместо того, чтобы объединять вместе части текста со ссылками, обозначающими маршрут, каждый маршрут в вашем приложении может быть представлен с помощью значения языка Haskell. Благодаря этому сразу исчезает большое количество ошибок 404 Not Found (404 Не найдено): просто невозможно получить неправильный URL. (Все еще можно сформировать URL, который приведет к ошибке 404, например, из-за ссылки на несуществующее сообщение в блоге. Тем не менее, все адреса будут сформированы правильно).

Так как же это магия работает? В каждом сайте есть тип данных `route`, и для каждого шаблона ресурсов есть свой собственный конструктор. Для нашего предыдущего примера мы получим что-то вроде следующего:

```
data MySiteRoute = HomepageR
                  | AddEntryR
                  | EntryR EntryId
```

Если вы хотите перейти по ссылке на домашнюю страницу, вы используете `HomepageR`. Чтобы разместить ссылку на конкретную запись, вы должны использовать конструктор `EntryR` с параметром `EntryId`. Например, чтобы создать новую запись и перейти на нее, вы могли бы написать:

```
entryId <- insert (Entry "My Entry" "Some content")
redirect RedirectTemporary (EntryR entryId)
```

Во всех языках Hamlet, Lucius и Julius есть встроенная поддержка таких типобезопасных адресов URL. Внутри шаблона на языке Hamlet вы можете легко создать ссылку на страницу с дополнительной записью:

```
<a href=@{AddEntryR}>Create a new entry.
```

Что же самое интересное? Точно также, как и в случае с сущностями `Persistent`, компилятор обеспечит вам полный порядок. Если вы поменяли какие-либо из ваших маршрутов (например, вы хотите включить в число ваших маршрутов год и месяц), Yesod заставит вас повсюду в коде обновить каждую ссылку.

Обработчики

После того, как вы определите ваши маршруты, вы должны сообщить Yesod, как вы хотите отвечать на запросы. Это то место, где в игру вступают функции - обработчики. Настройка очень проста: для каждого ресурса (например, `HomepageR`) и метода запроса, создается функция с именем `methodResourceR`. Для нашего предыдущего примера, нам потребовалось бы четыре функции: `getHomepageR`, `getAddEntryR`, `postAddEntryR` и `getEntryR`.

Все параметры, полученные из маршрута, передаются в качестве аргументов в функцию - обработчик. В функции `getEntryR` первый аргумент будет иметь тип `EntryId`, тогда как во всех остальных функциях никаких аргументов не будет вообще.

Функции - обработчики размещаются внутри монады `Handler`, в которой поддерживается большое количество возможностей, например, перенаправление запроса, доступ к сессии и выполнение запросов к базе данных. Что касается последней возможности, то типичный способ начать работу с функцией `getEntryR` будет выглядеть следующим образом:

```
getEntryR entryId = do
    entry <- runDB $ get404 entryId
```

Это позволит запустить действие, с помощью которого из базы данных будет получена запись, ассоциированная с данным ID. Если такой записи нет, то будет возвращен ответ 404.

Каждая функция - обработчик вернет некоторое значение, которое должно быть экземпляром типа `HasReps`. Это еще одна особенность, где свою роль играет RESTful: вместо того, чтобы просто вернуть некоторый фрагмент HTML или некоторый объект JSON, вы можете возвращать значение, которое может представлять собой и то, и другое в зависимости от заголовка запроса HTTP `Accept`. Другими словами, ресурс в Yesod является специфическим элементом данных, и его можно возвращать в одном из множества *представлений*.

Виджеты

Предположим, вы хотите добавить навигационную панель на нескольких различных страницах вашего сайта. Эта навигационная панель будет загружать пять самых последних постов блога (хранившихся в вашей базе данных), генерировать HTML, а затем использовать несколько фрагментов CSS и Javascript для соблюдения общего стиля сайта.

Если для того, чтобы собрать эти компоненты вместе, нет высокоуровневого интерфейса, то реализация может быть причиной головной боли. Вы можете добавить CSS к файлу CSS, который используется на всем сайте, но такое добавление вспомогательных деклараций не всегда является тем, что вам необходимо. Все тоже самое относится к Javascript, хотя с ним немного хуже: наличие дополнительного фрагмента Javascript может вызвать проблемы на странице, для работы с которой он не был предназначен. Вы также нарушаете модульность, поскольку вам потребуется генерировать результаты, получаемые из базы данных, с использованием нескольких обработчиков.

В Yesod, у нас есть очень простое решение: виджеты. Виджет является фрагментом кода, в котором воедино связаны HTML, CSS и Javascript, что позволяет вам добавлять содержимое сразу в заголовок и в тело веб страницы, и вы сможете запустить любой произвольный код, для которого есть обработчик. Например, для того, чтобы реализовать нашу навигационную панель:

```
-- Get last five blog posts. The "lift" says to run this code like we're in the handler.
entries <- lift $ runDB $ selectList [] [LimitTo 5, Desc EntryPosted]
toWidget [hamlet|
<ul .navbar>
    $forall entry <- entries
        <li>#{entryTitle entry}
    []
toWidget [lucius| .navbar { color: red } |]
toWidget [julius|alert("Some special Javascript to play with my navbar"); |]
```

Но здесь заложено даже больше, чем делается. Когда вы в Yesod создаете страницу, то стандартный подход состоит в объединении нескольких виджетов в один виджет, в котором содержится весь контент вашей страницы, а затем к нему применяется функция `defaultLayout`. Эта функция определена для каждого сайта, и представляет собой использование стандартного макета сайта.

сегда есть два подхода, определяющих, куда следует поместить код CSS и Javascript:

1. Объединить их и поместить их внутри вашего HTML в теги `style` и `script`, соответственно.
2. Поместить их во внешние файлы и обращаться к ним с помощью тегов `link` и `script`, соответственно.

Кроме того, размер вашего Javascript может быть автоматически уменьшен до минимума. Второй вариант является более предпочтительным, поскольку он позволяет выполнить несколько дополнительных оптимизаций:

1. Файлы создаются с именами, создаваемыми с помощью хэш-функции по содержимому файла. Это означает, что вы можете пользоваться ими и в будущем, причем не беспокоясь о том, что пользователи получать устаревший контент.
2. Ваш JavaScript можно загружать асинхронно.

Второй пункт требует некоторой доработки. Виджеты содержат не только JavaScript в его исходном виде, но в них также есть список Javascript-зависимостей. Например, во многих сайтах есть ссылки на библиотеку JQuery, а затем в них добавляются несколько фрагментов Javascript, в которых эта библиотека используется. Yesod может с помощью `yesod.js` автоматически преобразовать все это в асинхронную загрузку.

Другими словами, виджеты позволяют создавать модульный динамически компонуемый код, что в результате ведет к исключительно эффективному представлению ваших статических ресурсов.

Подсайты subsite

Во многих веб-сайтах есть общие области функциональных возможностей. Пожалуй, двумя наиболее распространенными примерами этого служат статические файлы и аутентификация. В Yesod, вы можете легко поместить такой код в виде подсайта `subsite`. Все, что вам нужно сделать, это добавить к вашим маршрутам дополнительную строку. Например, чтобы добавить статический подсайт, вы должны написать:

```
/static StaticR Static getStatic
```

Первый аргумент сообщает, откуда начинается подсайт. Для статического подсайта обычно используется `/static`, но вы можете использовать все, что вы захотите. `StaticR` является именем маршрута; оно также полностью зависит от вас, но, по соглашению, используется `StaticR`. `Static` – это имя статического подсайта, это единственное, что вы не можете поменять. `getStatic` является функцией, которая возвращает настройки статического сайта, например, где расположены статические файлы.

Как и все обработчики, у обработчиков подсайтов также есть доступ к функции `defaultLayout`. Это означает, что для хорошо продуманного подсайта будет автоматически использоваться внешний дизайн вашего сайта без каких-либо дополнительных вмешательств с вашей стороны.

22.6. Усвоенные уроки

Работа на проектом Yesod принесла очень много пользы. Она дала мне возможность работать над большими системами с различными группами разработчиков. Меня в действительности потрясло то, насколько конечный продукт отличается от того, что я первоначально намеревался сделать. Я начал работу на Yesod, составив список целей. В этом списке осталось совсем немного из тех основных функций, которые мы в настоящее время рекламируем в Yesod, и большая часть этого списка уже не та, что я планировал реализовать. Первый урок:

У вас будет более полное представление о системе, которая вам необходима, только после начала работы на ней. Не привязывайте себя к вашей первоначальной идее.

Поскольку это был мой первый крупный кусок кода на языке Haskell, я во время разработки Yesod узнал много нового о языке. Я уверен, что у многих может возникнуть чувство: «Как же я смог написать код, наподобие этого?». Даже при том, что исходный код был не такого калибра, как код Yesod, который есть у нас на данный момент, он был достаточно добротным с тем, чтобы стать толчком к росту проекта. Второй урок заключается в следующем:

Вас не должно отпугивать то, что у вас якобы нет достаточно мастерства в использовании инструментария, имеющегося у вас есть под рукой. Напишите настолько хороший код, насколько это возможно, а затем улучшайте его.

Одним из самых трудных шагов в разработке Yesod был переход от команды разработчиков, состоящей из одного человека — меня, к сотрудничеству с другими разработчиками. Все просто началось со сбора запросов, помещаемых на GitHub, и, в конце концов, закончилось появлением нескольких разработчиков, сопровождающих основной код. Я создал несколько своих собственных шаблонов разработки, которые нигде не были объяснены или документированы. В результате, участники проекта столкнулись с трудностями, когда захотели попробовать мои последние неописанные изменения. Это стало препятствием для многих других, кто хотел бы принять участие в проекте или протестировать его.

Когда Грег Вебер (Greg Weber) поднялся на борт в качестве еще одного ведущего проекта Yesod, он положил использовать много стандартов кодирования, которых катастрофически не хватало. Что еще усугубляло проблемы, это некоторые трудности, присущие экспериментированию с инструментальным набором разработчика на языке Haskell, а именно наличие большого количества пакетов, которые использовались в Yesod. С тех пор одной из целей всей команды разработчиков Yesod было создание стандартных сценариев и инструментов для автоматизации сборки проекта. Большинство из этих инструментов вернулись на своем пути развития обратно в исходное сообщество любителей языка Haskell. Последний урок состоит в следующем:

Сразу решайте, как сделать так, чтобы ваши проект был доступным для других.

[На главную](#) -> [MyLDP](#) -> [Тематический каталог](#) ->

23. Проект Yocto

Глава 23 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

Проект Yocto является проектом с открытым исходным кодом, выступающим в роли отправной точки для разработчиков встраиваемых систем на основе Linux в ходе создания специализированных дистрибутивов программных продуктов вне зависимости от применяемого для этого аппаратного обеспечения. Благодаря спонсорской помощи со стороны организации Linux Foundation, проект Yocto является системой, функционально превосходящей простые системы сборки. Он предоставляет в распоряжение разработчиков инструменты, процессы, шаблоны и методы для быстрого создания и внедрения продуктов в рамках рынка встраиваемых систем. Одним из ключевых компонентов проекта Yocto является система сборки Poky Build system. Так как Poky является большой и сложной системой, мы сфокусируем свое внимание на одном из ее ключевых компонентов с названием BitBake. BitBake является инструментом для сборки, созданным под впечатлением от системы Portage из дистрибутива Gentoo и используемым как проектом Yocto, так и сообществами OpenEmbedded для работы с метаданными и создания образов Linux-систем на основе их исходного кода.

В 2001 году компания Sharp Corporation представила КПК SL-5000 с названием Zaurus, работающий под управлением дистрибутива Linux для встраиваемых систем с названием Lineo. Спустя некоторое время после представления КПК Zaurus, Chris Larson основал проект OpenZaurus Project, направленный на создание дистрибутива на основе Linux для замены существующего в SharpROM и использующий систему сборки с названием buildroot. После создания проекта сторонние участники начали добавлять поддержку новых пакетов программного обеспечения наряду с дополнительными возможностями сборки дистрибутивов для других устройств незадолго до того, как система сборки проекта OpenZaurus начала давать сбои. В январе 2003 года в сообществе началось обсуждение возможности создания новой системы сборки, поддерживающей модель

функционирования сообщества в рамках стандартной системы сборки дистрибутивов на основе Linux для встраиваемых систем. В конечном счете это привело к созданию проекта OpenEmbedded. Chris Larson, Michael Lauer и Holger Shurig начали работу в рамках проекта OpenEmbedded с портирования сотен пакетов проекта OpenZaurus для работы с новой системой сборки.

Благодаря этой работе был создан проект Yocto. В основе проекта лежала система сборки Poky, созданная Richard Purdie. Проект начинал свое существование в виде стабилизированной ветки проекта OpenEmbedded, используя основной набор сотен рецептов проекта OpenEmbedded для сборки под ограниченное количество архитектур. Со временем проекты были объединены в нечто большее, чем система сборки программного обеспечения для встраиваемых систем и сформировали завершенную платформу для разработки программного обеспечения с плагином для среды разработки Eclipse, заменой инструмента fakeroot и возможностью работы с образами на основе QEMU. Примерно в ноябре 2010 года организация Linux Foundation выступила с заявлением о том, что работа над системой будет продолжаться в рамках проекта Yocto, получающего спонсорскую поддержку от Linux Foundation. После этого было установлено соглашение о том, что проекты Yocto и OpenEmbedded будут осуществлять координацию работы над ключевым набором метаданных пакетов, называемым OE-Core, комбинируя лучшие черты систем Poky и OpenEmbedded с увеличивающимися масштабами использования уровней для дополнительных компонентов.

23.1. Введение в систему сборки Poky Build System

Система сборки Poky является основой проекта Yocto. В рамках стандартной конфигурации Poky может предоставлять начальный образ системы, настраиваемый в диапазоне от минимального образа, предоставляющего возможность доступа с использованием командной оболочки, до совместимого со стандартом Linux Standard Base образа, использующего прототип пользовательского интерфейса с названием Sato на основе GNOME Mobile and Embedded (GMAE). При использовании этих основных типов образов, уровни метаданных могут быть добавлены для расширения функций; уровни позволяют создать дополнительный стек программного обеспечения для заданного типа образа, добавить в него пакеты для поддержки аппаратного обеспечения (board support packages - BSP) для беспроблемной работы с дополнительным аппаратным обеспечением или даже создать новый тип образа. Используя версию 1.1 системы Poky с кодовым названием "edison", мы покажем то, как BitBake использует рецепты и файлы конфигурации в ходе генерации образа для встраиваемой системы.

При высокогорневом анализе видно, что процесс сборки начинается с установки параметров окружения оболочки для последующей сборки. Для выполнения этой операции используются исходные данные из файла `oe-init-build-env`, который находится в корневой директории дерева исходного кода системы Poky. С помощью этого файла настраивается окружение оболочки, создается начальный изменяемый набор файлов конфигурации и осуществляется взаимодействие с окружением выполнения системы BitBake путем использования файла сценария, позволяющего Poky установить, выполняются ли минимальные системные требования.

Например, одним из наблюдаемых с помощью данного сценария параметров является наличие инструмента Pseudo, являющегося заменой `fakeroot`, переданной проекту Yocto компанией Wind River Systems. В этот момент сценарий `bitbake-core-image-minimal`, например, должен иметь возможность создать полнофункциональное окружение для кросскомпиляции, после чего сформировать образ Linux-системы на основе описания образа, соответствующего сценарию `core-image-minimal`, из исходного кода таким образом, как это описано на уровне метаданных проекта Yocto.

Высокоуровневый обзор процесса выполнения задачи Poky

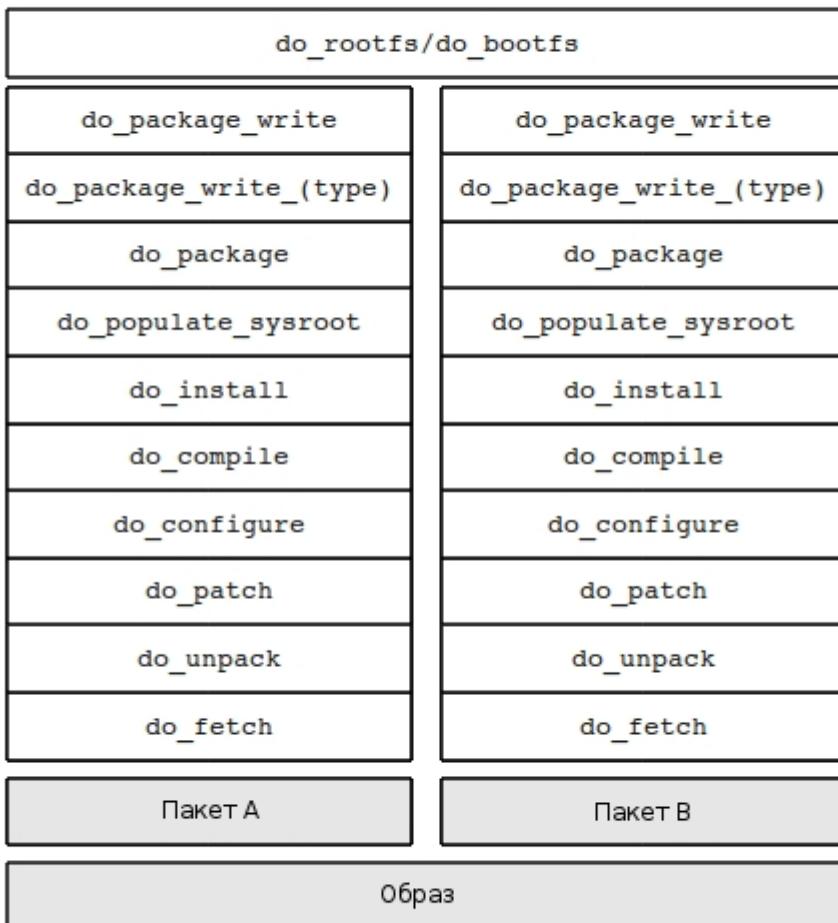


Рисунок 23.1: Высокоуровневый обзор процесса выполнения задачи Poky

В ходе формирования нашего образа BitBake произведет разбор файлов конфигурации, подключит любые дополнительно заданные уровни, классы, задачи или рецепты и начнет с создания цепочки зависимостей с приоритетами. Этот процесс позволяет создать упорядоченную карту задач с установлением их приоритетов. После этого BitBake будет использовать полученную карту задач для установления того, в каком порядке и какие пакеты должны быть собраны для наиболее оптимального разрешения зависимостей компиляции. Задачи, необходимые большинству других задач, имеют большие приоритеты и, следовательно, начинают работу раньше в ходе процесса сборки. Очередь выполнения задач для нашей сборки создана. BitBake также сохраняет итоги разбора метаданных и в том случае, если при последующих запусках устанавливается факт изменения метаданных, могут быть повторно разобраны только измененные метаданные. Планировщик и система разбора данных из состава BitBake являются одними из наиболее интересных архитектурных решений в рамках BitBake, а некоторые окружающие их решения вместе с их реализацией силами разработчиков BitBake также будут рассмотрены далее.

После этого BitBake осуществляет выполнение задач из цепочки, создавая программные потоки (количество которых ограничивается переменной `BB_NUMBER_THREADS` в файле `conf/local.conf`) для выполнения этих задач в предварительно заданном порядке. Выполняемые в ходе процесса сборки пакета задачи могут быть изменены, вставлены в начало или конец очереди с помощью со-

ответствующих рецептов. Основной, стандартный порядок выполнения задач сборки пакета начинается с получения и распаковки исходных кодов пакета, конфигурации и кросскомпиляции распакованного исходного кода. После этого скомпилированный исходный код разделяется на пакеты и над результатами компиляции проводятся различные действия, такие, как сбор отладочной информации для пакета. После этого разделенные пакеты упаковываются в пакеты поддерживаемого формата; поддерживаются форматы пакетов RPM, ipk и deb. После этого BitBake будет использовать эти пакеты для создания корневой файловой системы.

Концепции системы сборки Poky

Одной из наиболее мощных возможностей системы сборки Poky является то обстоятельство, что каждый аспект процесса сборки контролируется с помощью метаданных. Метаданные могут быть свободно разделены на группы файлов конфигурации или рецептов сборки пакетов. Рецепт сборки является набором неисполняющихся метаданных, используемых системой BitBake для установки значений переменных или указания дополнительных задач, выполняемых в процессе сборки. Рецепт сборки содержит такие поля, как описание рецепта, версия рецепта, лицензия пакета и адрес центрального репозитория исходного кода. Он также может указывать на то, что процесс сборки использует autotools, make, distutils или любой другой процесс сборки и в этом случае базовые функции могут быть установлены с помощью классов, унаследованных от класса уровня OE-Core, описанного в файлах директории `./meta/classes`. Дополнительные задачи также могут быть описаны наряду с условиями их выполнения. BitBake также поддерживает директивы `_prepend` и `_append` в качестве методов расширения функций задачи путем выполнения инъекции кода с добавлением суффиксов в начало и конец описания задачи.

Конфигурационные файлы могут быть разделены на два типа. Первый тип файлов предназначен для конфигурации системы BitBake и всего процесса сборки, а второй - для конфигурации различных уровней, используемых системой Poky для создания различных типов результирующего образа. Под уровнем понимается любая группа метаданных, позволяющая реализовывать какую-либо дополнительную функцию. Они могут использоваться для создания пакетов поддержки аппаратного обеспечения, предназначенных для новых устройств, создания дополнительных типов образов или добавления в образ дополнительного программного обеспечения, не обрабатываемого с помощью основных уровней. Фактически основной набор метаданных проекта Yocto с названием `meta-yocto` является уровнем, добавляемым выше уровня метаданных OE-Core с названием `meta`, с помощью которого добавляется дополнительное программное обеспечение и типы образов к заданным на уровне OE-Core.

Примером использования уровней может служить процесс создания образа для устройства NAS на основе платформы Intel n660 (Crownbay), использующего новый 32-битный ABI x32 для архитектуры x86-64, причем выбранное программное обеспечение для создания пользовательского интерфейса будет добавляться с помощью специального уровня.

Задавшись целью, мы разделим функции образа в соответствии с уровнями. На самом нижнем уровне мы используем уровень создания пакетов для поддержки аппаратного обеспечения с целью добавления в образ программных компонентов, предназначенных для поддержки специфичных для платформы Crownbay функций аппаратного обеспечения, например, видео-драйверов. Так как мы хотим использовать x32, нам придется использовать экспериментальный уровень `meta-x32`. Функции устройства NAS могут быть добавлены уровнем выше с помощью примерного уровня устройства NAS от проекта Yocto с названием `meta-baryon`. И наконец, мы используем воображаемый уровень с названием `meta-myproject` для добавления программного обеспечения и файлов конфигурации с целью создания графического пользовательского интерфейса, предназначенного для управления устройством NAS.

В ходе настройки окружения BitBake некоторые начальные файлы конфигурации генерируются на основе данных из пакета `oe-build-init-env`. Эти конфигурационные файлы позволяют нам в не-

которой степени контролировать то, как и какие файлы генерируются системой Poky. Первым файлом из набора этих конфигурационных файлов является файл `bblayers.conf`. Этот файл мы будем использовать для добавления дополнительных уровней при сборке нашего примера проекта.

Ниже приведено примерное содержание файла `bblayers.conf`:

```
# Значение LAYER_CONF_VERSION повышается при каждом несовместимом
# изменении файла build/conf/bblayers.conf
LCONF_VERSION = "4"
BBFILES ?= ""
BBLAYERS = " \
/home/eflanagan/poky/meta \
/home/eflanagan/poky/meta-yocto \
/home/eflanagan/poky/meta-intel/crownbay \
/home/eflanagan/poky/meta-x32 \
/home/eflanagan/poky/meta-baryon \
/home/eflanagan/poky/meta-myproject \
"
```

Файл описания уровней `bblayers.conf` содержит переменную `BBLAYERS`, которая используется BitBake для поиска уровней. Для лучшего понимания нам следует также рассмотреть устройство используемых уровней. Используя `meta-baryon` ([git://git.yoctoproject.org/meta-baryon](http://git.yoctoproject.org/meta-baryon)) в качестве примера уровня, поинтересуемся содержимым файла конфигурации уровня. Этот файл, `conf/layer.conf`, разбирается средствами BitBake после начального разбора файла `bblayers.conf`. Используя полученную информацию, BitBake добавляет дополнительные рецепты сборки, классы и файлы конфигурации в окружение сборки.

Распределение уровней в BitBake



Рисунок 23.2: Пример распределения уровней в BitBake

Ниже приведено содержание файла `layer.conf` из состава `meta-baryon`:

```
# Файл конфигурации уровня meta-baryon
# Copyright 2011 Intel Corporation
```

```
# Известна директория конфигурации, объединим путь к ней со значением переменной
BBPATH для предпочтительного использования наших версий
BBPATH := "${LAYERDIR}:${BBPATH}"

# Известны директории рецептов recipes-*, добавим пути к ним к значению переменной
BBFILES
BBFILES := "${BBFILES} ${LAYERDIR}/recipes-*/*/*.bb ${LAYERDIR}/recipes-
*/*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-baryon"
BBFILE_PATTERN_meta-baryon := "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-baryon = "7"
```

Все файлы конфигурации BitBake вносят вклад в процесс генерации хранилища данных BitBake, которое используется в процессе создания очереди выполнения задач. При начале сборки используется класс BitBake с названием `BBCooker`. Этот класс управляет выполнением задачи сборки путем выполнения действий (*baking*) в соответствии с рецептами (*recipes*). Одной из первых задач, выполняемой классом, является попытка загрузки и разбора данных конфигурации. Для информирования системы сборки о том, где она должна искать эти данные конфигурации (и в свою очередь о том, где искать метаданные рецептов), вызывается метод класса `parseConfigurationFiles`. При наличии нескольких исключений, первым конфигурационным файлом, который разыскивается рассматриваемым классом, является файл `bbplayers.conf`. После того, как заканчивается разбор данного файла, BitBake производит разбор файлов `layer.conf` для каждого из уровней.

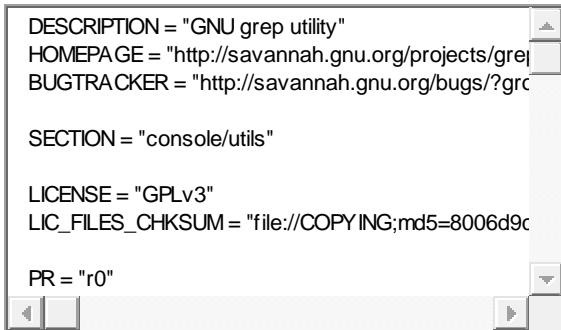
После окончания разбора файлов конфигурации уровней метод `parseConfigurationFiles` разбирает файл `bitbake.conf`, главной задачей которого является установка глобальных переменных времени сборки, таких, как переменные, отвечающие за структуру имен директорий для различных директорий корневой файловой системы (`rootfs`), а также переменные с начальным значением `LDFLAGS` для использования во время компиляции. Большинство конечных пользователей никогда не прибегнет к манипуляциям с этим файлом, так как практически все необходимые параметры, которые может быть необходимо изменить, будут находиться в рамках контекста рецепта сборки вместо файла для всей системы сборки или могут быть переопределены с помощью конфигурационного файла, такого, как `local.conf`.

Как только этот файл разобран, BitBake также подключает конфигурационные файлы, которые относятся к каждому из уровней, заданных переменной `BBLAYERS`, и добавляет найденные в этих файлах переменные в свое хранилище данных.

Ниже приведен фрагмент файла `bitbake.conf`, иллюстрирующий подключенные файлы конфигурации:

```
include conf/site.conf
include conf/auto.conf
include conf/local.conf
include conf/build/${BUILD_SYS}.conf
include conf/target/${TARGET_SYS}.conf
include conf/machine/${MACHINE}.conf
```

Пример рецепта BitBake для утилиты `grep`:



23.2. Архитектура BitBake

Перед тем, как мы перейдем к подробному рассмотрению части архитектурных решений, примененных в системе BitBake, полезно будет понять то, как BitBake собственно работает. Для того, чтобы в полной мере оценить прогресс развития системы BitBake, мы рассмотрим ее начальную версию, BitBake 1.0. В рамках этого первого релиза BitBake цепочка зависимостей для сборки формировалась на основе зависимостей рецептов. В случае какой-либо ошибки в процессе сборки образа, система BitBake должна была перейти к выполнению следующей задачи и позднее повторить ранее неудачную попытку сборки программного компонента. Очевидно, это означало то, что сборка занимала слишком много времени. Другая особенность работы системы BitBake заключалась в том, что каждая из переменных, используемых рецептами, хранилась в одном очень большом словаре. Учитывая количество рецептов, а также количество переменных и задач, необходимых для завершения сборки образа, можно сделать вывод, что система BitBake 1.0 требовала большого объема памяти для работы. В то время, когда оперативная память была достаточно дорогой и системы работали с меньшим ее объемом, сборки могли заканчиваться неудачами. Ситуация с исчерпанием оперативной памяти (и записью данных в раздел подкачки!) была неприемлема для системы, так как процесс сборки является длительным. В своем первозданном виде система хотя и выполняла поставленные перед ней задачи (иногда), но делала это очень медленно, потребляя чрезмерно большие количества ресурсов. Хуже того, в рамках версии 1.0 системы BitBake не было представлено концепции долговременного кэша данных или разделения данных состояния, а также не было возможности осуществлять инкрементальные сборки, поэтому в случае неудачи при сборке приходилось повторять ее с самого начала.

Краткий обзор различий между актуальной версией 1.13.3 системы BitBake, используемой в рамках системы сборки Poky "edison" и версией 1.0 указывает на появление реализации клиент-серверной архитектуры BitBake, долговременного кэша данных, хранилища данных, а также применение оптимизации путем использования техники копирования при записи данных в хранилище, реализации системы разделения данных состояния и значительных улучшений в алгоритмах формирования цепочек зависимостей для задач и пакетов. Данный эволюционный процесс привел к повышению стабильности, производительности и динамичности функционирования системы. Большая часть этих функций была продиктована необходимостью выполнения более быстрых и надежных процессов сборок с затратами меньшего количества ресурсов. Тремя усовершенствованиями системы BitBake, которые мы будем рассматривать, являются: реализация клиент-серверной архитектуры, оптимизация хранилища данных BitBake и работа по улучшению методов формирования цепочек зависимостей для сборок и задач в рамках BitBake.

Механизм межпроцессного взаимодействия системы BitBake

Так как мы уже достаточно хорошо знакомы с тем, как система сборки Poky использует файлы конфигурации, рецепты и уровни для создания образов встраиваемых систем, мы подготовлены к тому, чтобы взглянуть под капот системы BitBake и изучить метод объединения этих компонентов. Начиная с основного исполняемого файла системы BitBake, `bitbake/bin/bake`, мы можем приступить к рассмотрению процесса, выполняемого BitBake для настройки необходимой для начала сборки инфраструктуры. Первым интересующим нас элементом является механизм межпро-

цессного взаимодействия (Interprocess Communications - IPC) из состава BitBake. Изначально в рамках BitBake не было представлено концепции клиент-серверного взаимодействия. Эти функции были добавлены в BitBake спустя некоторое время для запуска множества процессов в рамках сборки, так как изначально система была однопоточной, а также для добавления альтернативных пользовательских возможностей.

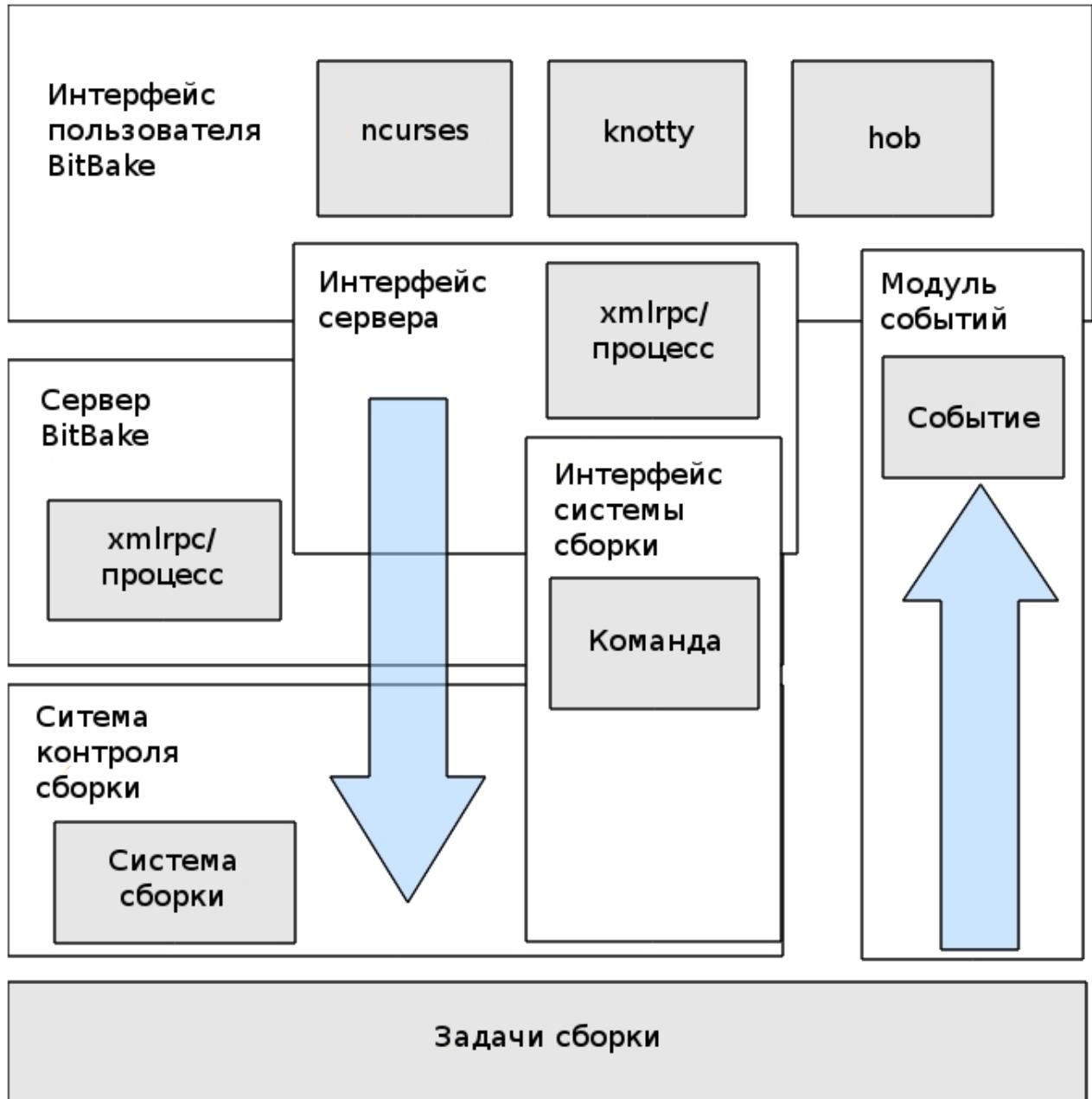


Рисунок 23.3: Обзор механизма межпроцессного взаимодействия системы BitBake

Все сборки, выполняемые с помощью системы Poky, начинаются с создания экземпляра пользовательского интерфейса. Пользовательский интерфейс предоставляет механизмы для записи данных событий, состояния и прогресса сборки наряду с механизмами приема событий выполнения задач сборки посредством модуля событий BitBake. Используемым по умолчанию пользовательским интерфейсом является интерфейс knotty, представляющий собой интерфейс командной строки сис-

темы BitBake. Он назван knotty или "(no)tty" из-за того, что работает как с tty-устройствами, так и с файлами, являясь одним из нескольких поддерживаемых интерфейсов. Одним из дополнительных пользовательских интерфейсов является интерфейс Hob. Hob является графическим интерфейсом для BitBake, напоминающим "BitBake commander". В дополнение к стандартным функциям, которые вы можете обнаружить в пользовательском интерфейсе интерфейсе knotty, Hob (разработанный Joshua Lock) предоставляет возможность модификации конфигурационных файлов, добавления дополнительных уровней и пакетов, а также полномасштабного изменения параметров процесса сборки.

Пользовательские интерфейсы системы BitBake имеют возможность отправлять команды следующему подключенному приложению BitBake модулю, реализующему функции сервера BitBake. Как и в случае с пользовательскими интерфейсами, BitBake также поддерживает множество различных типов серверов, таких, как XMLRPC. Стандартным сервером, который использует большинство пользователей при запуске системы BitBake с интерфейсом knotty, является сервер процесса BitBake. После запуска сервера приложение BitBake подключает модуль сборки.

Модуль сборки является основной частью BitBake, с помощью которой вызывается большая часть наиболее интересных процессов в ходе сборки с использованием системы Poky. Модуль сборки управляет разбором метаданных, инициирует генерацию деревьев зависимостей и задач, а также непосредственно управляет процессом сборки. Одной из функций архитектуры серверной части BitBake является предоставление различных возможностей раскрытия командного API для опосредованного доступа к нему со стороны пользовательского интерфейса. Модуль команд является рабочим элементом системы BitBake, осуществляющим запуск команд сборки и генерирующими события, которые передаются пользовательскому интерфейсу, минуя обработчик событий BitBake. В тот момент, когда приложение BitBake подключает модуль сборки, он инициализирует хранилище данных BitBake, после чего начинает разбор всех конфигурационных файлов системы сборки Poky. После этого модуль создает объект текущей очереди и начинает сборку.

Хранилище данных DataSmart системы BitBake с возможностью копирования при записи

В версии 1.0 системы BitBake переменные после извлечения из файлов помещались в один очень большой словарь в ходе инициализации класса данных. Как было сказано ранее, это приводило к проблемам, так как при работе с очень большими словарями языка Python операции записи и доступа к элементам осуществляются достаточно медленно и если на сборочной машине в процессе сборки закончится физическая память, будет использоваться раздел подкачки. Хотя такой сценарий и маловероятен для большинства систем в конце 2011 года, во время создания проекта OpenEmbedded и системы BitBake среднестатистические характеристики компьютеров обычно включали в себя объем оперативной памяти меньше одного или двух гигабайт.

Эта особенность являлась одним из слабых мест ранних версий системы BitBake. Двумя основными недостатками, которые требовали исправления для повышения производительности, являлись: во-первых, отсутствие возможности предварительного создания цепочки зависимостей сборки; во-вторых, необходимость сокращения объема данных, хранящихся в памяти. Большая часть хранящихся данных рецептов не изменялась от рецепта к рецепту; например, при наличии значений переменных TMPDIR, BB_NUMBER_THREADS и других глобальных переменных BitBake хранение в памяти всех данных окружения для каждого из рецептов являлось неэффективным. Решением проблемы оказался разработанный Tom Ansell словарь с поддержкой метода копирования при записи, который " злоупотреблял классами для того, чтобы быть замечательным и быстрым". Модуль с реализацией метода копирования при записи системы BitBake является одновременно и особо радикальным и разумным решением. Выполнение команды `python BitBake/lib/bb/COW.py` и исследование модуля облегчат ваше понимание принципа работы реализации механизма копирования при записи и способа использования этой реализации средствами системы BitBake для эффективного хранения данных.

Модуль DataSmart, использующий словарь с поддержкой метода копирования при записи, хранит все данные начальной конфигурации системы Poky, данные из файлов с расширениями `.conf` и `.bbclass` в словаре в виде объектов данных. Каждый из этих объектов может содержать другой объект данных с информацией исключительно об отличиях данных. Таким образом, в том случае, если с помощью рецепта происходит изменение данных начальной конфигурации, вместо копирования всей конфигурации с целью локализации, на уровне ниже в стеке данных, подвергающихся копированию при записи, сохраняется объект, содержащий информацию об отличиях данных родительского объекта. При попытке доступа к переменной модуль данных будет использовать модуль DataSmart для объектов на верхнем уровне стека. В том случае, если переменная не обнаруживается, осуществляется переход на нижний уровень стека до тех пор, пока переменная не обнаруживается, либо генерируется ошибка.

Одна из других интересных особенностей модуля DataSmart находится в области раскрытия переменных. Так как переменные BitBake могут содержать исполняемый код на языке Python, одной из необходимых для выполнения операций является передача значения переменной методу `bb.codeparser` для установления того, что это значение представляет собой корректный код на языке Python и не содержит циклических ссылок. В качестве примера переменной, содержащей код на языке Python, может использоваться фрагмент файла

```
./meta/conf/distro/include/tclibc-eglibc.inc:
```

```
LIBCEXTENSION = "${@[" -gnu"][(d.getVar('ABIEXTENSION', True) or "") != ""]}"
```

Эта переменная подключается с помощью одного из конфигурационных файлов уровня OE-Core с именем `./meta/conf/distro/include/defaultsetup.conf` и используется для формирования набора стандартных параметров в различных конфигурациях дистрибутивов, которые могут использоваться при работе с Poky и OpenEmbedded. Этот файл позволяет импортировать некоторые специфичные для библиотеки `eglibc` переменные, значения которых устанавливаются в зависимости от значения другой переменной BitBake с именем `ABIEXTENSION`. В процессе создания хранилища данных код на языке Python из данной переменной должен быть разобран и проверен для того, чтобы избежать неудачного завершения задач, использующих эту переменную.

Планировщик BitBake

После того, как система BitBake произвела разбор файлов конфигурации и создала хранилище данных, ей необходимо произвести разбор рецептов, требующихся для создания образа, и сформировать цепочку сборки. Эта операция является одним из наиболее существенных усовершенствований системы BitBake. Изначально система BitBake получала приоритеты сборки из рецепта. Если в рамках рецепта была задана переменная `DEPENDS`, предпринималась попытка установления того, какие программные компоненты следует собрать для того, чтобы выполнить заданные этой переменной требования. В том случае, если выполнение задачи завершалось неудачей из-за того, что для сборки не хватало какого-либо предварительного условия, задача просто убиралась для последующего повторного выполнения. Этот подход имел очевидные недостатки, связанные и с производительностью, и с надежностью.

Так как предварительно формируемой цепочки зависимостей не создавалось, порядок выполнения задач устанавливался непосредственно во время сборки. Это обстоятельство ограничивало возможности системы BitBake однопоточным режимом сборки. Для того, чтобы продемонстрировать, насколько непроизводительной может оказаться сборка образов с помощью BitBake в однопоточном режиме, следует упомянуть о том, что при сборке образа самого малого размера "core-image-minimal" на стандартной машине разработчика в 2011 году (Intel Core i7 с 16 ГБ оперативной памяти DDR3) потребуется около трех или четырех часов для сборки полного набора инструментов кросскомпиляции и применения их для создания пакетов, которые впоследствии будут использоваться для создания образа. Для сравнения, сборка на той же машине с переменными

BB_NUMBER_THREADS со значением 14 и PARALLEL_MAKE со значением "-j 12" занимает от 30 до 40 минут. Как можно представить, работа в однопоточном режиме без предварительного формирования последовательности выполнения задач с использованием более медленного аппаратного обеспечения с меньшим объемом оперативной памяти, большое количество которой может быть занято копиями всего хранилища данных, потребует значительного большего времени.

Зависимости

При разговоре о зависимостях сборки нам следует проводить разделение зависимостей различных типов. Зависимости сборки, задаваемые с помощью переменной DEPENDS, являются чем-либо, что нам требуется предварительно предоставить для того, чтобы сборочная система Poky смогла собрать требуемый пакет, в то время, как зависимости времени исполнения, задаваемые с помощью переменной RDEPENDS, требуют от образа установки заданных с помощью переменной RDEPENDS пакетов наряду с запрашиваемым пакетом. Возьмем, например, пакет с названием `task-core-boot`. Если мы рассмотрим рецепт этого пакета, расположенный по пути

```
meta/recipes-core/tasks/task-core-boot.bb
```

мы обнаружим две установленные переменные BitBake: RDEPENDS и DEPENDS. Система BitBake использует эти два поля в процессе создания цепочки зависимостей.

Ниже приведен фрагмент файла `task-core-boot.bb`, демонстрирующий использование переменных DEPENDS и RDEPENDS:

```
DEPENDS = "virtual/kernel"
...
RDEPENDS_task-core-boot = "\\\nbase-files \\\nbase-passwd \\\nbusybox \\\ninitscripts \\\n..."
```

Пакеты не являются единственными элементами, зависимости которых отслеживаются средствами BitBake. Задачи также имеют свои зависимости. В рамках очереди выполнения сборки BitBake мы выделяем четыре типа задач: внутренне-зависимые задачи, зависимые на основе значения переменной DEPENDS задачи, зависимые на основе значения переменной RDEPENDS задачи, а также зависимые от задач других пакетов задачи.

Внутренне-зависимые задачи устанавливаются в рамках рецепта и позволяют добавить задачу перед и/или после другой задачи. Например, мы можем добавить задачу с названием `do_deploy` в рецепт путем добавления строки `addtask deploy before do_build after do_compile`. Эта строка позволит добавить зависимость для запуска задачи `do_deploy` перед запуском задачи `do_build`, но после завершения выполнения задачи `do_compile`. Зависящие от значений переменных DEPENDS и RDEPENDS задачи являются задачами, выполняющимися после обозначенной задачи. Например, если мы хотим выполнить задачу `do_deploy` для пакета после выполнения задачи `do_install` для пакетов, заданных переменными DEPENDS или RDEPENDS, наш рецепт будет включать строку `do_deploy[deptask] = 'do_install'` или `do_deploy[rdeptask] = 'do_install'`. В случае задач, зависимых от задач других пакетов, если мы хотим чтобы заданная задача зависела от задачи другого пакета, мы добавим строку, изменив приведенный выше пример использования функции `do_deploy` следующим образом: `do_deploy[depends] = "<название целевого пакета>:do_install"`.

Очередь выполнения сборки

Так как в процессе сборки образа могут быть задействованы тысячи рецептов, каждый из которых может содержать множество пакетов и задач со своими зависимостями, на данный момент BitBake пытается разобраться в этих зависимостях и сформировать какую-либо структуру, которую можно будет использовать для установления порядка выполнения задач. После того, как модуль сборки получает в процессе инициализации объекта `bb.data` полный список пакетов, которые необходимо собрать, он приступает к созданию структуры распределения задач с приоритетами на основе этих данных для формирования упорядоченного списка необходимых для выполнения задач под названием "очередь выполнения сборки" (*runqueue*). Сразу же после формирования очереди выполнения сборки BitBake может начать выполнение находящихся в ней задач с учетом их приоритетов, причем каждая задача будет выполняться в отдельном потоке.

При задействовании модуля установления источника пакета, BitBake в первую очередь проверяет, установлено ли значение переменной `PREFERRED_PROVIDER` для заданного пакета или образа. В том случае, если более чем один рецепт может предоставить заданный пакет и так как задачи устанавливаются в рамках рецептов, от BitBake необходимо принять решение, какой источник пакета будет использован. Система отсортирует все источники пакета, поставив в соответствие каждому из них приоритет, установленный на основе совокупности различных критериев. Например, предпочтительные версии программного обеспечения будут иметь больший приоритет, чем все остальные. Однако, BitBake также учитывает версию пакета наряду с его зависимостями от других пакетов. После того как выбран рецепт, который будет использован для создания пакета, BitBake последовательно исследует значения переменных `DEPENDS` и `RDEPENDS` данного рецепта и перейдет к установлению источников для полученных названий пакетов. В ходе этой цепной реакции формируется список пакетов, необходимых для генерации образа, а также списки источников для данных пакетов.

Теперь очередь выполнения сборки располагает полным списком пакетов, которые должны быть собраны, а также цепочкой зависимостей. Для начала работы модуль выполнения сборки должен создать объект `TaskData`, таким образом начав сортировку структуры распределения задач на основе приоритетов. Этот процесс начинается с рассмотрения каждого найденного предназначенно для сборки пакета, разделения задач, необходимых для генерации этого пакета и присвоения каждой из этих задач приоритета на основании количества пакетов, требующих ее. Задачи с более высоким приоритетом имеют большее количество зависимостей и, следовательно, в целом выполняются раньше в процессе сборки. После завершения этой работы модуль очереди выполнения сборки подготавливает данные для преобразования объекта `TaskData` непосредственно в очередь выполнения сборки.

Процесс формирования очереди выполнения сборки отчасти сложен. Вначале BitBake обходит список имен задач объекта `TaskData` для установления зависимостей задач. Так как выполняется обработка данных объекта `TaskData`, начинается создание структуры распределения задач с приоритетами. После окончания этого процесса в случае отсутствия циклических зависимостей, задач, выполнение которых невозможно, а также других подобных проблем, распределение задач будет упорядочено на основании приоритетов и модулю выполнения сборки будет возвращен объект с полной очередью выполнения сборки. Модуль выполнения сборки предпримет попытку последовательного выполнения задач из очереди. В зависимости от размера образа и вычислительных ресурсов, системе сборки Poky может потребоваться от получаса до нескольких часов для генерации набора инструментов кросскомпиляции при указании пакета и выборе необходимого образа встраиваемой системы на основе Linux. Стоит отметить, что с момента выполнения команды `bitbake <имя_образа>` с использованием командной строки, весь процесс, начинающийся с выполнения задач из очереди выполнения сборки, занимает меньше нескольких секунд.

23.3. Заключение

После моих дискуссий с членами сообщества и личных исследований, я установила несколько областей, в которых некоторые вещи, возможно, должны были быть реализованы иначе, а также усвоила несколько ценных уроков. Важно отметить, что оценка десятилетней разработки не участвующим в ней человеком не является критикой тех, кто вложил свое время и усилия в весь этот замечательный набор программного обеспечения. Говоря от лица разработчиков, наиболее сложной частью нашей работы является прогнозирование того, что нам понадобится спустя годы и как мы могли бы спроектировать фреймворк, чтобы эти возможности работали прямо сейчас. Без некоторых проблем это удается только единицам.

Первый усвоенный урок заключается в том, что нужно быть уверенными в необходимости разработки соответствующей стандартом документации с четкими формулировками, которая понятна сообществу. Она должна проектироваться с учетом максимальной гибкости и последующего развития.

Одной из областей, где я лично столкнулась с недоработкой документации является моя работа над классом создания отметки лицензии из состава OE-Core, а в особенности данная недоработка проявилась при работе с переменной LICENSE. Так как не существовало четко документированной стандартизации того, что может содержать переменная LICENSE, обзор множества доступных рецептов показал значительные различия в объявлениях. Различные строки, являющиеся значениями переменной LICENSE, содержали все что угодно, от значений, использующих абстрактно-синтаксические деревья Python в строковом представлении, до значений, вероятность извлечения полезных данных из которых была крайне мала. Существовало соглашение, которое обычно использовалось в сообществе; однако, это соглашение предполагало различные варианты, некоторые из которых были менее корректными, чем другие. Эта проблема не была вызвана действиями разработчиков, которые создавали рецепт; она была вызвана неспособностью сообщества выработать стандарт.

Хотя небольшая предварительная работа со значением переменной LICENSE помимо проверки ее существования все таки была проведена, о стандартизации значений данной переменной никто не позаботился. Большую часть проблем удалось бы избежать в случае заблаговременной разработки поддерживаемого всеми стандарта в масштабе проекта.

Следующий усвоенный урок является более общим и относится к недоработке, наблюданной не только в рамках проекта Yocto, но и в других крупномасштабных проектах, находящихся в зависимости от архитектуры системы. Одна из наиболее важных идей для разработчиков, позволяющая ограничить трудозатраты по копированию, рефакторингу и удалению кода, с необходимостью которых они могут столкнуться при работе над проектом, формулируется следующим образом: тратьте время - много времени - на проектирование интерфейсов и проработку архитектурных решений.

Если вы думаете, что потратили достаточно времени на работу над архитектурой, вероятно, это не так. Если вы думаете, что потратили недостаточно времени на работу над архитектурой, то это несомненно так и есть. Затраты большего количества времени на проектирование интерфейса не усложнят последующие операции по удалению кода или даже реализации значительных архитектурных изменений, но, несомненно, сократят объем повторной разработки в долгосрочной перспективе.

Проектируйте свое программное обеспечение так, чтобы оно было настолько модульным, насколько это возможно, ведь рано или поздно вы будете возвращаться к некоторым участкам кода для проведения любых операций начиная с небольших исправлений и заканчивая повторной разработкой, а когда вам все таки придется работать с кодом, его замена станет менее сложной.

Очевидной областью, в которой такой подход мог помочь проекту Yocto, является установление потребностей конечных пользователей, эксплуатирующих системы с малым объемом оперативной

памяти. В том случае, если бы реализация хранилища данных BitBake была заранее более тщательно продумана, возможно, нам удалось бы спрогнозировать вероятность появления проблем, связанных с тем, что хранилище данных занимает большой объем памяти, и заранее доработать его.

Данный урок заключается в том, что хотя и практически невозможно предугадать каждую из проблем, с которой столкнется ваш проект в период своего существования, выделение времени для серьезного планирования его интерфейса может помочь в снижении последующих трудозатрат. Проекты BitBake, OE-Core и Yocto являются удачными в этом плане, так как большое количество работы над их архитектурой было произведено на ранних стадиях развития. Это обстоятельство позволило нам произвести значительные изменения архитектуры проекта без больших сложностей и ущерба проекту.

23.4. Благодарности

Во-первых, благодарю Chris Larson, Michael Lauer и Holger Schurig, а также многих других людей, которые вносили свой вклад в развитие BitBake, OpenEmbedded, OE-Core и проекта Yocto в течение многих лет. Также благодарю Richard Purdie за то, что он предоставил в мое распоряжение свой мозг и помог разобраться как в исторических, так и в технических аспектах OE, а также за его постоянную поддержку и руководство, особенно в случаях исследования некоторых магических аспектов BitBake.

24.1. Приложение или библиотека

ØMQ является библиотекой, а не сервером обмена сообщениями. На это у нас пошло несколько лет работы с протоколом AMQP: на попытку стандартизировать в финансовой индустрии сетевой протокол для обмена бизнес сообщениями, на написание эталонной реализации для него и участие в нескольких крупномасштабных проектах, базирующихся в значительной степени на технологии обмена сообщениями, на понимание того, что что-то не так с классической клиент/серверной моделью умного сервера обмена сообщениями (брокера) и бессловесными клиентами обмена сообщениями.

Наша главная задача в то время была связана с производительностью: Если в середине находится сервер, то каждое сообщение должно пройти в сети два раза (от отправителя к брокеру и от брокера к приемнику), порождая при этом проблемы, связанные с задержками и пропускной способностью. Более того, если все сообщения передаются через брокера, то в какой-то момент он обязан стать узким местом.

Вторая задача была связана с развертыванием системы в крупномасштабных сетях: когда при развертывании пересекаются организационные границы, то концепция центрального органа управления всем потоком сообщений перестает применяться. Ни одна из компаний не готова уступить контроль сервером другой компании, есть коммерческая тайна и есть юридическая ответственность. Результат на практике состоит в том, что в компании есть один сервер обмена сообщениями с рукописными мостами для подключения к системам обмена сообщениями в других компаниях. Поэтому экосистема в целом сильно фрагментирована, а поддержка большого количества мостов для каждой участвующей компании не делает ситуацию лучше. Чтобы решить эту проблему, нам нужна полностью распределенная архитектура, архитектура, в которой каждым компонентом может управлять, возможно, иной хозяйствующий субъект. Учитывая, что блоком управления в серверной архитектуре является сервер, мы можем решить эту проблему путем установки отдельного сервера для каждого компонента. В таком случае можно дополнительно оптимизировать проект, сделав так, сервер и компонент будут использовать одни и те же процессы. Так, что в итоге у нас все это завершилось созданием библиотеки обмена сообщениями.

Проект ØMQ был начат, когда мы получили представление о том, как сделать работу с сообщениями без центрального сервера. Это требует переворот всей концепции сообщений с ног на голову и замены модели автономного централизованного хранилища сообщений в центре сети на архитектуру «умные конечные точки, молчащая сеть», которая базируется на принципе соединений «точка-точка». Техническим следствием этого решения было то, что проект ØMQ с самого начала был библиотекой, а не приложением.

Одновременно мы смогли доказать, что такая архитектура является более эффективной (меньшие задержки, более высокая пропускная способность) и более гибкой (она проще для построения различных сложных топологий, чем привязка к классической модели «ось и спицы»).

Однако, одним из непреднамеренных последствий было то, что выбор в пользу библиотечной модели улучшил удобство работы с продуктом. Снова и снова пользователи выражают свое удовольствие по поводу того, что им не требуется устанавливать автономный сервер обмена сообщениями и им управлять. Оказывается, что отсутствие сервера является предпочтительным вариантом, поскольку это сокращает эксплуатационные затраты (не требуется администратор сервера сообщений) и становится проще выход на рынок (нет необходимости вести переговоры о необходимости запуска сервера с клиентом, командой, осуществляющей управление, или эксплуатационной командой).

Усвоенный урок в том, что при запуске нового проекта вы должны отдавать предпочтение созданию библиотек, если это вообще возможно. Довольно легко создать приложение из библиотеки, запустив ее из тривиальной программы, но практически невозможно создать библиотеку из существующих исполняемых модулей. Библиотека для пользователей будет гораздо более гибкой и в то же время не потребует от них нетривиальных административных усилий.

24.2. Глобальное состояние

Глобальные переменные не всегда хороши при использовании с библиотеками. Библиотека может загружаться в процесс неоднократно, но даже в этом случае будет только один набор глобальных переменных. На рис.24.1 показана библиотека ØMQ, которая используется из двух различных и независимых библиотек. Затем приложение использует обе эти библиотеки.

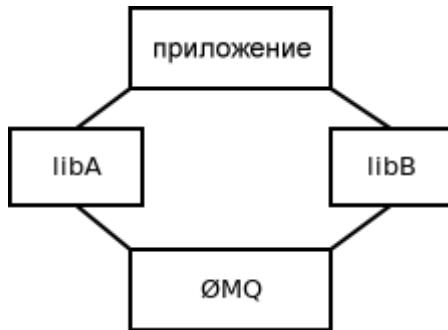


Рис.24.1: ØMQ используется двумя различными библиотеками

Когда возникает такая ситуация, то оба экземпляра ØMQ имеют доступ к одним и тем же переменным, в результате чего возникают состояния гонки, странные сбои и неопределенное поведение.

Чтобы избежать этой проблемы, в библиотеке ØMQ отсутствуют глобальные переменные. Вместо этого пользователь библиотеки ответственен за работу в явном виде с глобальным состоянием. Объект, содержащий глобальные состояния, называется *контекстом*. Хотя с точки зрения пользователя контекст выглядит более или менее похожим на пул рабочих потоков, с точки зрения ØMQ это просто объект для хранения любого глобального состояния, которое нам понадобится. На ри-

сунке, приведенном выше, библиотека `libA` должна иметь свой собственный контекст точно также, как и библиотека `libB`. Тогда ни одна из них не сможет вывести из строя или повлиять на другую библиотеку.

Усвоенный здесь урок довольно очевиден: не используйте в библиотеках глобальное состояние. Если вы это делаете, то в случае, когда в одном и том же процессе будет использовано два экземпляра библиотеки, библиотека выйдет из строя.

24.3. Производительность

Когда проект `ØMQ` был запущен, его основной целью была оптимизация производительности. Производительность системы обмена сообщениями оценивается с помощью двух метрик: пропускной способности - сколько сообщений может быть передано в течение определенного количества времени, и задержкой - сколько времени требуется сообщению для того, чтобы добраться от одной конечной точки к другой.

На какой показатель мы должны ориентироваться? Какая связь между ними? Разве это не очевидно? Запустите тест, разделите общее время теста на количество пришедших сообщений и вы получаете задержку. Разделите количество сообщений на время и вы получаете пропускную способность. Другими словами, задержка обратно пропорциональна величине пропускной способности. Тривиально, не так ли?

Вместо того, чтобы сразу начать кодирование, мы потратили несколько недель на более подробное исследование метрик производительности и выяснили, что отношения между пропускной способностью и задержкой гораздо более тонкое, чем приведенное выше, и часто метрики довольно нелогичны.

Представьте себе, А отправляет сообщения для В (смотрите рис.24.2.) Общее время теста составляет 6 секунд. Было передано 5 сообщений. Следовательно пропускная способность равна 0,83 сообщения/сек ($5/6$), а задержка равна 1,2 сек ($6/5$), не так ли?

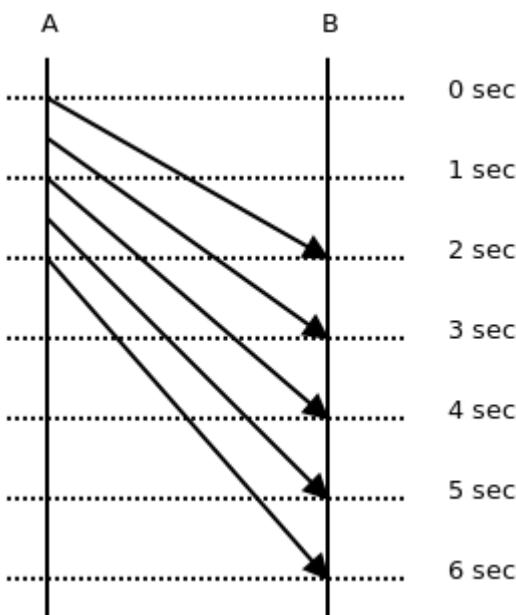


Рис.24.2: Отправка сообщений из А в В

Снова взгляните на диаграмму. На ней видно различное время для каждого сообщения, которое поступает из А в В: 2 сек, 2,5 сек, 3 сек, 3,5 сек, 4 сек. В среднем это 3 секунды, что довольно да-

леко от нашего первоначального расчета в 1,2 секунды. Этот пример показывает заблуждения, которые интуитивно делаются относительно метрик производительности.

Теперь взгляните на пропускную способность. Общее время теста составляет 6 секунд. Однако, в А будет затрачено всего 2 секунды для того, чтобы отправить все сообщения. С точки зрения А пропускная способность равна 2,5 сообщений/сек ($5/2$). В затрачивается 4 секунды для того, чтобы принять все сообщения. Так что с точки зрения В пропускная способность равна 1,25 сообщений/сек ($5/4$). Ни одно из этих значений не соответствует нашему первоначальному расчету в 1,2 сообщений/сек.

Короче говоря, задержка и пропускная способность являются двумя различными метриками - это очевидно. Важно понимать различие между ними и их взаимосвязь. Задержку можно измерить только между двумя различными точками в системе; нет такого понятия, как задержка в точке А. Каждое сообщение имеет свою собственную задержку. Вы можете вычислить среднее значение задержек нескольких сообщений, однако, нет такого понятия, как задержка потока сообщений.

Пропускная способность, с другой стороны, может измеряться только в одной точке системы. Имеется пропускная способность отправителя, пропускная способность принимающей стороны, есть пропускная способность любой промежуточной точкой между ними, но нет такого понятия, как общая пропускная способность всей системы. И пропускная способность имеет смысл только для набора сообщений, нет такого понятия, как пропускная способность одного сообщения.

Что касается отношений между пропускной способности и задержкой, то, оказывается, действительно, между ними есть взаимосвязь; однако, в формуле есть интегралы и мы здесь ее обсуждать не будем. Для получения дополнительной информации, читайте литературу по теории очередей.

При оценке производительности системы обмена сообщениями есть еще очень много подводных камней, так что мы не будем вдаваться в подробности. Упор нужно сделать на следующем усвоенном уроке: Убедитесь, что вы понимаете проблему, которую вы решаете. Даже такая проблема, как просто «сделать что-то более быстрым» может для ее правильного понимания потребовать большого объема работы. Более того, если вы не понимаете проблему, вы, вероятно, на основе невидных предположений и популярных мифов создадите код, в результате чего это решение будет иметь недостатки или, по крайней мере, будет гораздо более сложным или гораздо менее полезным, чем это можно было бы сделать.

24.4. Критический путь

Мы в процессе оптимизации обнаружили, что на производительность оказывают решающее влияние следующие три фактора:

- Количество операций выделения памяти
- Количество системных вызовов
- Модель распараллеливания

Однако не каждая операция выделения памяти и не каждый системный вызов оказывает одинаковое влияние на производительность. Характеристикой, которая нас интересует в системах обмена сообщениями, является количество сообщений, которое мы можем передать между двумя конечными точками в течение определенного количества времени. Кроме того, нас может интересовать то, как долго сообщение передается из одной точки в другую.

Но, учитывая то, что ØMQ предназначена для сценариев с долгоживущими соединениями, время, необходимое для установления соединения или времени, необходимое для обработки ошибки соединения, в основном, значения не имеет. Эти события происходят очень редко, и поэтому их влияние на общую производительность незначительно.

Та часть кода, которая снова и снова используется очень часто, называется *критическим путем* (*critical path*); оптимизировать следует критический путь.

Давайте рассмотрим пример: библиотека ØMQ не очень оптимизирована относительно выделения памяти. Например, при обработке строк, она часто выделяет новую строку для каждого промежуточного этапа преобразования. Тем не менее, если мы посмотрим строго на критический путь, т. е. фактическую передачу сообщений, мы увидим, что в библиотеке практически не происходит выделение памяти. Если сообщения небольшие, то это всего лишь одно выделение памяти на 256 сообщений (эти сообщения хранятся в одном большом выделенном участке памяти). Если, кроме того, поток сообщений устойчив и без огромных пиков трафика, то количество выделений памяти на критическом пути падает до нуля (выделенные участки памяти не возвращаются, но снова и снова используются повторно).

Усвоенный урок: Есть разница в том, где выполнять оптимизацию. Оптимизация фрагментов кода, которые не находятся на критическом пути, является напрасной тратой усилий.

24.5. Выделение памяти

Если предположить, что вся инфраструктура инициализирована и соединение между двумя конечными точками уже установлено, есть только одно, для чего выделяется память, это - само сообщение. Таким образом, для оптимизации критического пути мы должны были изучить, как происходит выделение памяти под сообщения и как сообщения передаются вверх и вниз по стеку.

В сфере высокопроизводительных сетей общеизвестно, что лучшая производительность достигается за счет четкого баланса между стоимостью выделения памяти под сообщение и стоимостью копирования сообщения (например, <http://hal.inria.fr/docs/00/29/28/31/PDF/Open-MX-IOAT.pdf>: сравните различные подходы для «малых», «средних» и «больших» сообщений). Для небольших сообщений, копирование гораздо дешевле, чем выделение памяти. Имеет смысл вообще не выделять никаких новых кусков памяти, а вместо этого по мере необходимости копировать сообщение в заранее выделенную память. Для больших сообщений, с другой стороны, копирование гораздо дороже, чем выделение памяти. Имеет смысл один раз выделить память для сообщения и, вместо копирования данных, передать указатель на выделенный блок. Такой подход называется «нулевым копированием».

ØMQ обрабатывает оба варианта прозрачно. Сообщение ØMQ представлено структурой, в которой спрятаны детали. Содержимое очень маленьких сообщений хранится непосредственно в структуре. Таким образом, при создании копии структуры действительно копируется данные сообщения. Когда сообщение станет большим, для него выделяется отдельный буфер, а в структуре хранится только указатель на буфер. Создание копии структуры не приводит к копированию данных сообщения, что разумно в случае, когда размер сообщения равен мегабайтам (рис. 24.3). Следует отметить, что в последнем случае в буфере подсчитывается количество указателей, так что можно использовать указатели из нескольких структур и не требуется копировать данные.

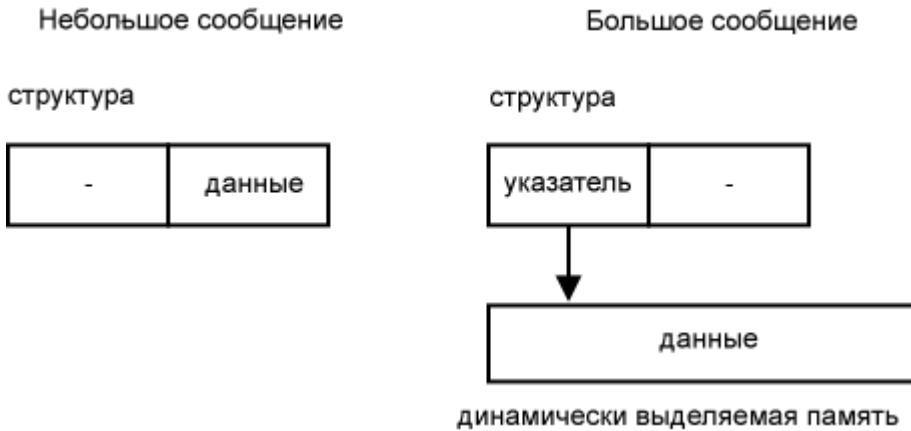


Рис.24.3: Копирование сообщения (или только указателя)

Усвоенный урок: Когда вы думаете о производительности, то не считайте, что наилучшее решение только одно. Может случиться, что есть несколько подклассов проблемы (например, небольшие сообщения и большие сообщения), для каждого из которых есть свой собственный оптимальный алгоритм.

24.6. Пакетная обработка

Как уже было упомянуто, огромное количество системных вызовов в системе обмена сообщениями может привести к возникновению узких мест по производительности. На самом деле, проблема гораздо более общая. Возникают очень нетривиальные потери производительности, связанные с обходом стека вызовов и, следовательно, при создании высокопроизводительных приложений разумно настолько, насколько это возможно, избегать выполнение обхода стека вызовов.

Рассмотрим рис.24.4. Чтобы отправить четыре сообщения, вы должны четыре раза пройти весь сетевой стек целиком (т.е. ØMQ, glibc, границу пользовательского пространства/пространства ядра, реализацию TCP, реализацию IP, слой Ethernet, сам NIC и снова вернуться).

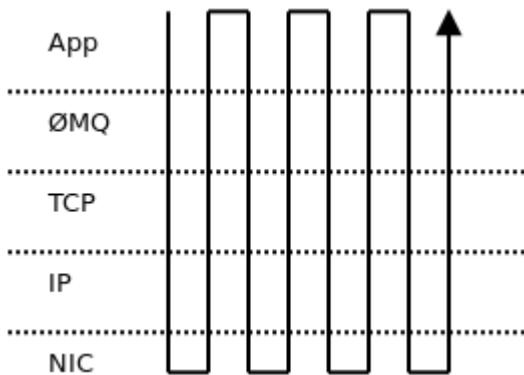


Рис.24.4: Отправка четырех сообщений

Тем не менее, если вы решите объединить эти четыре сообщения в один пакет, то потребуется только один обход стека (рис. 24.5). Влияние на пропускную способность сообщений может быть огромным: до двух порядков, особенно если сообщения маленькие и сотни таких сообщений можно упаковывать в один пакет.

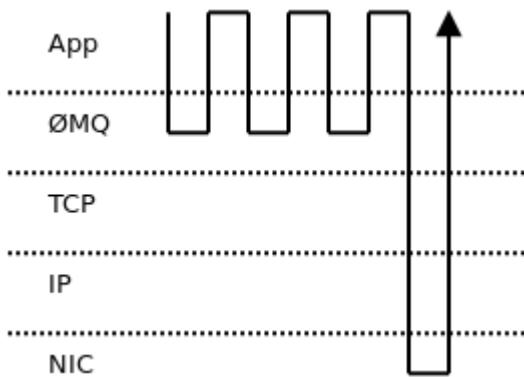


Рис.24.5: Пакетная обработка сообщений

С другой стороны, пакетная обработка может иметь негативное влияние на задержку. Возьмем, например, хорошо известный алгоритм Нэйгла (Nagle), который реализован в TCP. Он задерживает исходящие сообщения в течение определенного количества времени и объединяет все накопленные данные в одном пакете. Очевидно, что полная задержка первого сообщения в пакете гораздо больше, чем задержка последнего. Поэтому обычно, чтобы в приложениях снизить задержку, алгоритм Нэйгла отключается. Обычно отключается даже пакетная обработка на всех уровнях стека (например, возможность объединения прерываний NIC).

Но опять же, отсутствие пакетной обработки не означает больших перемещений по стеку и не приводит к низкой пропускной способности сообщений. Мы, кажется, столкнулись с дилеммой между пропускной способностью и задержкой.

Библиотека ØMQ пытается обеспечить сравнительно низкие задержки в сочетании с высокой пропускной способностью за счет использования следующей стратегии: когда поток сообщений не-большой и не превышает пропускную способность сетевого стека, ØMQ отключает всю пакетную обработку с тем, чтобы улучшить задержку. Компромисс здесь в несколько большем использовании ЦП - мы все еще должны часто проходить через стек. Однако в большинстве случаев это не является проблемой.

Когда скорость сообщений превышает пропускную способность сетевого стека, сообщения должны быть поставлены в очередь и храниться в памяти до тех пор, пока стек не будет готов принять их. Очередь означает, что задержка будет расти. Если сообщение находится в очереди одну секунду, то полная задержка будет равна, по меньшей мере, одну секунду. Что еще хуже, поскольку размер очереди растет, задержка будет постепенно увеличиваться. Если размер очереди не ограничен, то задержка может быть больше любого заранее заданного предела.

Было обнаружено, что даже если сетевой стек настроен на минимально возможную задержку (выключен алгоритм Нэйгла, выключено объединение прерываний NIC и т.д.) задержка все еще может быть достаточно большой из-за эффекта очереди, описанного выше.

В такой ситуации имеет смысл агрессивно начинать использование пакетной обработки. Нет ничего, чтобы можно было потерять, поскольку в любом случае задержка и так уже высока. С другой стороны, агрессивное использование пакетной обработки увеличивает пропускную способность и может убрать из очереди ожидающие сообщения, что в свою очередь означает, что задержка будет постепенно падать поскольку задержка из-за очереди уменьшается. Как только в очереди станет мало сообщений, пакетная обработка может быть отключена для еще большего снижения задержки.

Еще одно наблюдение состоит в том, что пакетная обработка должна выполняться только на самом верхнем уровне. Если сообщения группируются там, нижние слои так или иначе не имеют

никакого отношения к пакетной обработке, поскольку все алгоритмы пакетной обработки ничего не делают, кроме как вводят дополнительную задержку.

Усвоенный урок: Для того, чтобы в асинхронной системе получить оптимальную пропускную способность в сочетании с оптимальным временем ответа, выключите все алгоритмы пакетной обработки на низких уровнях стека и включите пакетную обработку на самом верхнем уровне. Пакетная обработка требуется только тогда, когда новые данные прибывают быстрее, чем они могут быть обработаны.

24.7. Общий обзор архитектуры

До этого момента мы сосредоточили внимание на общих принципах, которые делают библиотеку ØMQ быстрой. Теперь мы взглянем на реальную архитектуру системы (рис. 24.6).

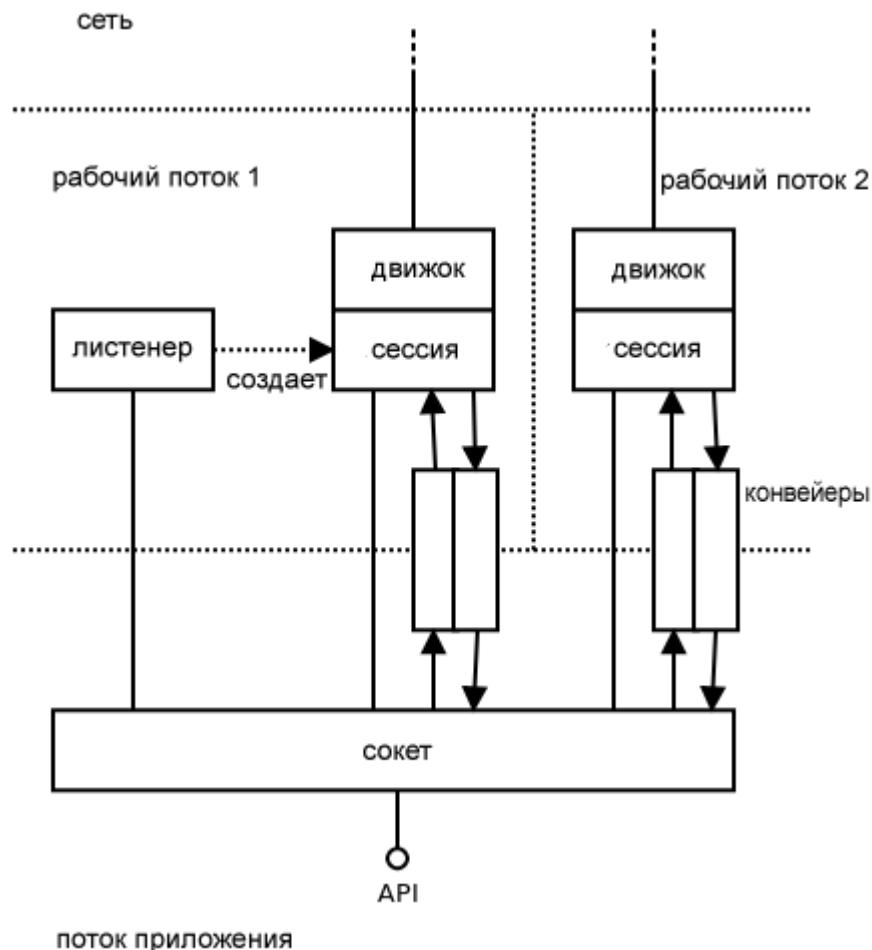


Рис.24.6: Архитектура ØMQ

Пользователь взаимодействует с ØMQ с использованием так называемых «сокетов». Они очень похожи на сокеты TCP, основное отличие в том, что каждый сокет может обрабатывать соединения с несколькими абонентами, что немного похоже на то, как это делают несвязанные сокеты UDP.

Объект сокета живет в потоке пользователя (смотрите обсуждение потоковых моделей в следующем разделе). Кроме этого, ØMQ работает в нескольких рабочих потоках, которые обрабатывают асинхронную часть соединения: чтение данных из сети, помещение сообщений в очередь, прием поступающих соединений и т.д.

Существуют различные объекты, находящиеся в рабочих потоках. Каждый из этих объектов принадлежит точно только одному родительскому объекту (принадлежность на диаграмме обозначается простой сплошной линией). Родительский объект может находиться в потоке, отличном от потока потомка. Большинство объектов принадлежат непосредственно сокетам; однако, есть несколько случаев, когда объект находится в собственности объекта, который принадлежит сокету. Все, что мы получаем, это дерево объектов, причем по одному такому дереву на сокет. Дерево используется в ходе завершения работы с сокетами; работа ни с одним из объектов не может быть завершена прежде, чем будет завершена работа со всеми его потомками. Таким образом, мы можем гарантировать, что процесс завершения будет работать так, как ожидается, например, ожидающие исходящие сообщения будут отправлены в сеть до завершения процесса отправки.

Грубо говоря, есть два вида асинхронных объектов; есть объекты, которые не участвуют в передаче сообщений, и есть объекты, которые участвуют. Объекты первого вида связаны главным образом с управлением соединением. Например, объект слушателя TCP (листенер) прослушивает входящие соединения TCP и создает объекты движка/сессии для каждого нового соединения. Аналогичным образом объект коннектора TCP (соединение) пытается подключиться к порту TCP и, в случае успеха, создает объект движка/сессии для управления подключением. Если соединение было разорвано, то объект коннектора пытается восстановить его (реконнектор).

Объекты второго вида представляют собой объекты, которые непосредственно участвуют в передаче данных. Эти объекты состоят из двух частей: *сессионный объект* (*session object*) отвечает за взаимодействие с сокетом ØMQ, а *объект движка* (*engine object*) отвечает за связь с сетью. Имеется только один вид сессионного объекта, но для каждого протокола, который поддерживается в ØMQ, имеются различные типы объектов движков. Т.е., у нас есть движки TCP, движки IPC (межпроцессное взаимодействие), движки PGM (надежный мультикастовый протокол —смотрите RFC 3208) и др. Набор движков можно расширять - в будущем мы можем выбрать для реализации, скажем, движок WebSocket или движок SCTP.

Сессии являются сессиями обмена сообщениями с сокетами. Есть два направления для передачи сообщений и каждое направление обрабатывается при помощи конвейерного объекта. Каждый конвейер является в своей основе очередью без блокировок, оптимизированной для быстрого прохождения сообщений между потоками.

Наконец, есть объект контекста (о нем рассказывалось в предыдущих разделах, но он не показан на рисунке), в котором хранится глобальное состояние и он доступен всем сокетам и всем асинхронным объектам.

24.8. Модель распараллеливания

Одним из требований для ØMQ было возможность использования многоядерных устройств; другими словами, чтобы можно было масштабировать пропускную линейно с увеличением числа доступных ядер процессора.

Наш предыдущий опыт работы с системами обмена сообщениями показал, что с использованием нескольких потоков в классическом пути (критические секции, семафоры и т.д.) не дает значительного улучшения производительности. В самом деле, многопоточная версия системы обмена сообщениями может быть более медленной, чем однопоточная, даже если измеренная осуществляется на многоядерном устройстве. Отдельные потоки просто тратят слишком много времени на ожидание друг друга, и в то же время требуют большого количества переключений контекста, что замедляет работу системы.

Учитывая эти проблемы, мы решили перейти на другую модель. Цель состояла в том, чтобы полностью избежать блокировок и позволить каждому потоку работать на полной скорости. Взаимодействие между потоками было реализовано с помощью асинхронных сообщений (событий), ко-

торые передаются между потоками. Это, как знают инсайдеры, является классической моделью актера (*actor model*).

Идея заключалась в том, чтобы запускать на каждом ядре процессора по одному рабочему потоку — наличие двух потоков, совместно использующих то же самое ядро, будет означать лишь большое количество переключений контекста без получения особых преимуществ. Каждый внутренний объект ØMQ, такой, как, скажем, движок TCP, будет тесно связан с конкретным рабочим потоком. Это, в свою очередь, означает, что нет никакой необходимости в критических секциях, взаимоисключаемых событиях (mutexes), семафорах и тому подобном. Кроме того, эти объекты ØMQ не будут перераспределяться между ядрами процессора, так что удастся избежать негативного влияния на производительность, связанного с загрязнением кэша (рис.24.7).



Рис.24.7: Несколько рабочих потоков

Благодаря такой конструкции исчезает много традиционных многопоточных проблем. Тем не менее, есть необходимость в том, чтобы рабочим потоком могли пользоваться множество объектов, что в свою очередь означает, что должен быть какой-то вид кооперативной многозадачности. Это означает, что нам нужен планировщик; объекты должны управляться при помощи событий, не надо реализовывать управление циклом всех событий; мы должны учесть возможность возникновения событий в произвольной последовательности, причем даже очень редких; мы должны обеспечить, чтобы ни один объект не удерживал процессор слишком долго и т.д.

Короче говоря, вся система должна стать полностью асинхронной. Никакой объект не должен выполнять блокирующих операций, поскольку он заблокирует не только себя, но также и все другие объекты, использующие тот же самый рабочий поток. Все объекты должны представлять собой, прямо или косвенно, автоматы или машины состояний. При наличии сотен или тысяч машин состояний, работающих параллельно, вы должны обеспечить все возможные взаимодействия между ними и, самое главное, правильно реализовать процесс завершения их работы.

Получается, что завершение работы полностью асинхронной системы является в чистом виде устрашающе сложной задачей. При попытке завершить работу тысячи движущихся частей, некоторые из которых работают, некоторые находятся в состоянии ожидания, некоторые - в процессе инициализации, некоторые из них уже завершили свою работу самостоятельно, возможны возникновения всех видов состояний гонки, утечки ресурсов и тому подобное. Подсистема завершения работы является, безусловно, самой сложной частью ØMQ. Быстрый просмотр трекера ошибок показывает, что около 30 - 50% обнаруженных ошибок связаны в той или иной форме с этапом завершения работы системы.

Усвоенный урок: Когда стремитесь к экстремальной производительности и масштабируемости, то рассмотрите модель актера; это чуть ли не единственная варианта в подобных случаях. Однако, если вы не пользуетесь специализированной системой, например, Erlang или самой ØMQ, вам придется написать и вручную отладить инфраструктуру большого объема. Кроме того, с самого начала подумайте о процедуре завершения работы системы. Это будет самая сложная часть кода, и если у вас нет четкого представления о том, как ее реализовать, вам, вероятно, следует в первую очередь пересмотреть использование модели актера.

24.9. Неблокирующие алгоритмы

В последнее время в моде стали неблокирующие алгоритмы. Это простые механизмы межпотокового взаимодействия, в которых не используются предоставляемые ядром примитивы синхронизации, такие как взаимоисключающие события или семафоры; они предпочитают выполнять синхронизацию с использованием атомарных операций процессора, таких как атомное сравнение и своп (CAS). Следует иметь в виду, что они не в буквальном смысле работают без блокировок — в действительности блокировки происходят за кулисами на аппаратном уровне.

ØMQ использует неблокирующую очередь в конвейерных объектах для передачи сообщений между потоками пользователя и рабочими потоками ØMQ. Есть два интересных аспекта, касающихся того, как ØMQ использует неблокирующую очередь.

Во-первых, в каждой очереди есть ровно один поток, который осуществляется запись, и ровно один поток, который осуществляет чтение. Если необходима связь типа «1-N», то создается несколько очередей (рис.24.8). Благодаря такой организации очереди не надо беспокоиться о синхронизации записи (есть только один поток, осуществляющий запись) или чтения (есть только один поток, осуществляющий чтение), причем очередь может быть реализована очень эффективным способом.



Рис.24.8: Очереди

Во-вторых, мы поняли, что хотя неблокирующие алгоритмы более эффективны, чем классические алгоритмы, базирующиеся на использовании взаимоисключаемых событий, атомарные операции процессора все еще достаточно дороги (особенно когда есть рассогласованность между ядрами процесса) и выполнение атомарной операции для каждого сообщения, которое записывается или читается, происходит гораздо медленнее, чем нам это могло подойти.

Способ ускорения — опять же - потоковая обработка. Представьте, что у вас есть 10 сообщений, которые должны быть записаны в очередь. Это может произойти, например, когда вы получили сетевой пакет, содержащий 10 небольших сообщений. Получение пакета является атомарным событием, вы не можете получить половину пакета. В результате этого атомарного события в неблокирующую очередь потребуется записать 10 сообщений. Нет никакого смысла выполнять атомарную операцию для каждого сообщения. Вместо этого, вы можете накопить сообщения в виде порции «предзаписи» в той части очереди, которая доступна исключительно для записывающего потока, а затем сбросить ее в очередь с помощью одной атомарной операции.

То же самое относится и к считыванию из очереди. Представьте себе 10 сообщений, рассмотренных выше, которые уже были помещены в очередь. Поток, осуществляющий чтение, может извлекать каждое сообщение из очереди с использованием атомарной операции. Тем не менее, это перебор; вместо этого, поток может с помощью одной атомарной операции перенести все ожидающие сообщения в порцию «предварительного чтения» в очереди. После этого, он может выбирать сообщения из буфера «предварительного чтения» по одному. Порция «предварительного чтения» принадлежит и доступна исключительно потоку, осуществляющему чтение, и, таким образом, на этой фазе вообще не нужна какая-либо синхронизация.

Стрелка в левой части рисунка 24.9 показывает, как можно с помощью модификации одного указателя выполнить сброс в очередь содержимого буфера предварительной записи. Стрелка в правой части показывает, как можно ничего не делая, а только изменив еще один указатель, сдвинуть содержимое очереди в буфер предварительного чтения.

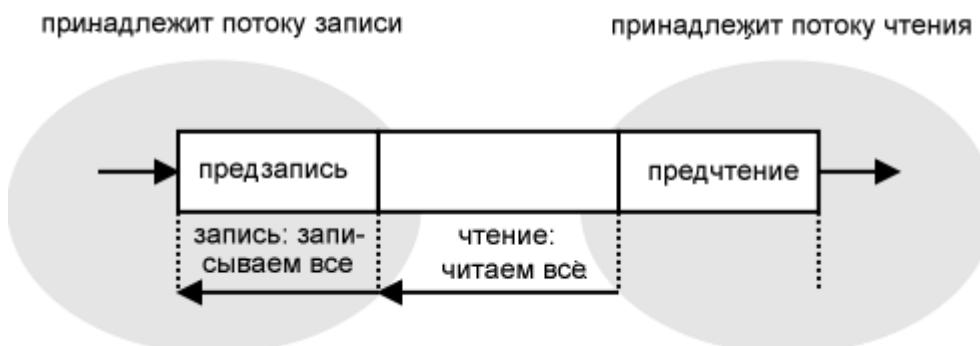


Рис.24.9: Неблокирующая очередь

Усвоенный урок: неблокирующие алгоритмы трудно придумывать, сложно реализовывать и почти невозможно отлаживать. Если возможно, то используйте существующие проверенные алгоритмы, а не изобретайте свои собственные. Если требуется экстремальная производительность, то не следует полагаться исключительно на неблокирующие алгоритмы. Хотя они и быстрые, производительность можно значительно улучшить, если поверх их сделать умную пакетную обработку.

24.10. Интерфейс API

Пользовательский интерфейс является наиболее важной частью любого продукта. Это единственная часть вашей программы, которая видна внешнему миру, и если вы сделаете ее неправильно, то мир будет ненавидеть вас. В конечном изделии это либо графический интерфейс, либо интерфейс командной строки. В библиотеках - это интерфейс API.

В ранних версиях ØMQ интерфейс API базировался на модели обменов и очередей AMQP (смотрите [спецификации AMQP](#)). С исторической точки зрения было бы интересно взглянуть на документ [white paper from 2007](#), в котором делается попытка примирить AMQP с бесброкерной моделью обмена сообщениями. Я потратил окончание 2009 года на его переписывание почти с нуля, чтобы вместо этого использовать интерфейс BSD Socket API. Это был поворотный момент; с этого момента количество применений библиотеки ØMQ взлетело. Если раньше это был нишевый продукт, используемый только горсткой экспертов по сообщениям, то после этого он стал обычным инструментом, удобным для любого. Через год или около того размер сообщества увеличился в десять раз, было реализовано несколько привязок к 20 различным языкам и т.д.

Пользовательский интерфейс определяет восприятие продукта. При практически оставшихся тех же самых функциональных возможностях - просто за счет изменения интерфейса API - ØMQ превратился из продукта уровня «enterprise messaging» (промышленной системы обмена сообщениями) в «сетевой» продукт. Иными словами, восприятие изменилось со «сложной части инфраструктуры

для крупных банков» на то, что «это помогает мне отправить мое сообщение длиной в 10 байт из приложения А в приложение В».

Усвоенный урок: Разберитесь с тем, каким, как вы хотите, должен быть ваш проект, и проектируйте соответствующий пользовательский интерфейс. Если у вас есть пользовательский интерфейс, который не совпадает с видением проекта, то это является 100% гарантией движения к провалу.

Одним из важных аспектов перехода на интерфейс BSD Sockets API было то, что он не был революционным недавно изобретенным API, а уже существовал и был хорошо известен. На самом деле, интерфейс BSD Sockets API является одним из старейших API, которым сегодня по-прежнему активно пользуются; он восходит к 1983 году и к 4.2BSD Unix. Он широко распространен и стабилен в течение буквально десятилетий.

Вышеуказанный факт дает много преимуществ. Во-первых, это интерфейс API, который известен каждому, поэтому кривая обучения будет до неприличия плоской. Даже если вы никогда не слышали о ØMQ, вы можете создать свое первое приложение через пару минут благодаря тому, что вы можете воспользоваться знаниями о BSD Sockets.

Во-вторых, использование широко распространенного интерфейса API позволяет интегрировать ØMQ с существующими технологиями. Например, представление объектов ØMQ в виде «сокетов» или «дескрипторов файлов» позволяет в одном и том же цикле событий выполнять обработку событий TCP, UDP, конвейеров, файлов и ØMQ. Другой пример: [экспериментальный проект](#) по переносу функций, похожих на ØMQ, в ядро Linux оказывается в реализации довольно простым. За счет применения того же самого по концептуальности фреймворка уже сейчас можно пользоваться большей частью инфраструктуры ØMQ.

Третьим и, вероятно, самым главным, является тот факт, что поскольку интерфейс BSD Sockets API выжил в течение почти трех десятилетий несмотря на многочисленные попытки его заменить, это означает, что в нем самом изначально есть нечто. Разработчики BSD Sockets API, намеренно или случайно, приняли правильные проектные решения. Используя это API, мы можем автоматически воспользоваться этими проектными решениями даже не зная, какими они были и какие проблемы они решали.

Усвоенный урок: Хотя интерес к повторному использованию кода был повышен с незапамятных времен, а позже к нему присоединилось повторное использование шаблонов, важно думать о повторном использовании решений в еще более общем виде. Когда разрабатываете продукт, взгляните на аналогичные продукты. Проверьте, какие не удались, а какие оказались успешными; изучите успешные проекты. Не поддавайтесь синдрому «Здесь ничего не придумано». Повторно используйте идеи, интерфейсы API, концептуальные фреймворки, которые вы посчитаете подходящими. Поступая таким образом, вы позволяете пользователям повторно использовать имеющиеся у них знания. В то же время вы можете избежать технических ловушек даже в случае, если вы на данный момент их не осознаете.

24.11. Шаблоны обмена сообщениями

В любой системе обмена сообщениями, наиболее важной проблемой проекта является то, каким способом пользователю приходится указывать какие сообщения направляются и в каком направлении. Существуют два основных подхода, и я считаю, что такая дилемма вполне универсальна и применима практически для любой проблемы, возникающей в области программного обеспечения.

Один из подходов заключается в принятии философии Unix «делать одно и делать это хорошо». Это означает, что область проблемы должна быть искусственно ограничена небольшой и хорошо понятной частью. Затем программа должна решить эту ограниченную задачу правильно и исчер-

пывающее. Примером такого подхода в сфере обмена сообщениями является [MQTT](#). Это протокол распределенных сообщений для группы потребителей. Он не может использоваться ни для чего другого (скажем, для RPC), но он прост в использовании и хорошо работает с распределенными сообщениями.

В другом подходе мы ориентируемся на общность и предоставляем мощную и глубоко настраиваемую систему. Примером такой системы является AMQP. Его модель очередей и обменов сообщениями предоставляет пользователям средства для программного определения практически любого алгоритма маршрутизации, который они могут придумать. Компромиссом, конечно, является множество параметров, о которых нужно позаботиться.

В ØMQ выбрана первая модель, поскольку в ней предполагается, что полученным продуктом сможет пользоваться в основном каждый, тогда как обобщенная модель для того, чтобы ей пользоваться, требует экспертов в области обмена сообщений. Чтобы это продемонстрировать, давайте посмотрим, как модель влияет на сложность API. Далее приведена реализация клиентской части RPC, построенная поверх обобщенной системы (AMQP):

```
connect ("192.168.0.111")
exchange.declare (exchange="requests", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
exchange.declare (exchange="replies", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
reply-queue = queue.declare (queue="", passive=false, durable=false,
    exclusive=true, auto-delete=true, no-wait=false, arguments={})
queue.bind (queue=reply-queue, exchange="replies",
    routing-key=reply-queue)
queue.consume (queue=reply-queue, consumer-tag="", no-local=false,
    no-ack=false, exclusive=true, no-wait=true, arguments={})
request = new-message ("Hello World!")
request.reply-to = reply-queue
request.correlation-id = generate-unique-id ()
basic.publish (exchange="requests", routing-key="my-service",
    mandatory=true, immediate=false)
reply = get-message ()
```

С другой стороны, ØMQ разбивает ландшафт обмена сообщениями на так называемые «шаблоны сообщений». Примерами шаблонов являются «публикация/подписка», «запрос/ответ» или «распараллеливаемый конвейер». Каждый шаблон обмена сообщениями полностью ортогонален другим шаблонам и может рассматриваться как отдельный инструмент.

Далее приведена еще одна реализация вышеприведенного приложения, выполненная в ØMQ с использованием шаблона «запрос/ответ». Обратите внимание на то, что все настройки сводятся к одному шагу выбора правильного шаблона обмена сообщениями («REQ»):

```
s = socket (REQ)
s.connect ("tcp://192.168.0.111:5555")
s.send ("Hello World!")
reply = s.recv ()
```

До этого момента мы утверждали, что конкретные решения лучше, чем обобщенные. Мы хотим, чтобы наши решения, чтобы быть как можно более конкретными. Тем не менее, мы одновременно хотим, насколько это окажется возможным, предложить нашим клиентам максимально широкий спектр функциональных возможностей. Как мы можем решить это кажущееся противоречие?

Ответ состоит из двух шагов:

1. Определение слоя стека, который касается конкретной проблемной области (например, транспорт, маршрутизация, презентация и т.д.).

2. Предоставление нескольких реализаций слоя. Это должны быть непересекающиеся реализации, отдельные для каждого варианта использования.

Давайте посмотрим на пример транспортного уровня стека Internet. Он предназначен для предоставления таких сервисов, как передача потоков данных, управления потоками, обеспечение надежности передачи и т.д., которые реализуются поверх сетевого уровня (IP). Он реализуется в соответствие с определением нескольких непересекающихся решений: TCP для соединений, ориентированных на надежную передачу потока, UDP для соединения с ненадежной пакетной передачей, SCTP для передачи нескольких потоков, DCCP для ненадежных соединений и так далее.

Обратите внимание, что каждая реализация полностью ортогональна: конечная точка UDP не может обмениваться сообщениями с конечной точкой TCP. А конечная точка SCTP не может обмениваться сообщениями с конечной точкой DCCP. Это означает, что в любой момент к стеку могут быть добавлены новые реализации без влияния на существующие части стека. И наоборот, о неудачных реализациях можно будет забыть и можно будет выбросить их без ущерба для жизнеспособности транспортного уровня в целом.

Тот же принцип относится и к шаблонам обмена сообщениями так, как это сделано в ØMQ. Шаблоны обмена сообщениями формируют слой (так называемый «слой масштабируемости») поверх транспортного уровня (TCP и аналоги). Реализациями этого слоя являются индивидуальные шаблоны обмена сообщениями. Они строго ортогональны — конечная точка «публикации/подписки» не может обмениваться сообщениями с конечной точкой «запроса/ответа» и т.д. Строгое разделение между шаблонами в свою очередь означает, что по мере необходимости могут быть добавлены новые шаблоны и что неудачные эксперименты с новыми шаблонами не повредят существующим шаблонам.

Усвоенный урок: Когда решается сложная и многогранная проблема, то может оказаться, что монолитное обобщенное решение может оказаться не лучшим. Вместо этого, мы можем рассмотреть проблемную область в виде абстрактного слоя и предложить несколько его реализаций, каждая из которых направлена на вполне конкретные условия использования. Когда вы так поступаете, то очень тщательно определите условия использования. Проверьте, что попадает в эту область, а что - нет. Из-за слишком агрессивного ограничения области использования, применение программного обеспечения может быть ограничено. Но если вы определите проблему слишком широко, продукт может стать слишком сложным, нечетким и запутанным для пользователей.

24.12. Заключение

Поскольку наш мир заселяется большим количеством маленьких компьютеров, подключаемых через Интернет - мобильными телефонами, считывателями меток RFID, планшетами и ноутбуками, устройствами GPS и т. д. - проблема распределенных вычислений перестает быть областью академической науки и становится повседневной проблемой, которую решает каждый разработчик. Решения, к сожалению, в основном являются предметно-ориентированными хакерскими трюками. Эта статья подытоживает наш опыт построения крупномасштабной распределенной системы на систематической основе. Она фокусируется на проблемах, которые представляют интерес с точки зрения архитектуры программ, и мы надеемся, что разработчики и программисты из сообщества сторонников открытого кода посчитают ее полезной.

24. Система обмена сообщениями ZeroMQ

Глава 24 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 2.

ØMQ является системой обмена сообщениями, или, с вашего позволения, «программным обеспечением среднего слоя, ориентированным на работу с сообщениями». Оно используется в разнооб-

разных средах, например, в финансовых сервисах, в разработке игр, во встраиваемых системах, в научных исследованиях и в аэрокосмической отрасли.

Системы обмена сообщениями работают в основном также, как и средства мгновенного обмена сообщениями в приложениях. В приложении принимается решение передавать событие в другое приложение (или в несколько приложений), оно собирает данные для отправки, высвечивает кнопку «send» («Отправить») и — вперед, обо всем остальном позаботится система обмена сообщениями.

В отличие от систем мгновенных сообщений (*instant messaging*), в системах обмена сообщениями (*messaging system*) нет графического интерфейса и в конечных точках не предполагается вмешательства человека в случае, если что-то пойдет не так. Поэтому системы обмена сообщениями должны быть как отказоустойчивыми, так и более быстрыми, чем обычные системы обмена мгновенными сообщениями.

Система ØMQ был изначально задумана как ультра-быстрая система сообщений для торговли акциями и поэтому основное внимание было уделено исключительно оптимизации. Первый год реализации проекта был посвящен разработке методики тестирования производительности и попытке определить архитектуру, которая была бы как можно более эффективной.

Позднее, примерно в течение второго года разработки, акцент сместился на создание общей системы для построения распределенных приложений и поддержке произвольных шаблонов сообщений, различных механизмов, транспортов, произвольных языковых сборок и т.д.

В течение третьего года акцент делался, главным образом, на улучшение удобства использования и обеспечении большей плавности кривой обучения. Мы взяли BSD Sockets API, пытаясь навести порядок в семантике отдельных шаблонов обмена сообщениями, и так далее.

Будем надеяться, что настоящая глава даст понимание того, как три цели, указанные выше, были преобразованы во внутреннюю архитектуру системы ØMQ, и даст несколько советов тем, кто борется с аналогичными проблемами.

Начиная со своего третьего года разработки ØMQ перерос работу только с кодом; есть инициатива стандартизации протоколов соединений, которые он использует, и экспериментальной реализации ØMQ как системы обмена сообщениями внутри ядра Linux и т.д. Эти темы не рассматриваются в данной книге. Тем не менее, для получения дополнительной информации вы можете обратиться к следующим интернет-ресурсам: <http://www.250bpm.com/concepts>, <http://groups.google.com/group/sp-discuss-group> и <http://www.250bpm.com/hits>.

24.1. Приложение или библиотека

ØMQ является библиотекой, а не сервером обмена сообщениями. На это у нас пошло несколько лет работы с протоколом AMQP: на попытку стандартизировать в финансовой индустрии сетевой протокол для обмена бизнес сообщениями, на написание эталонной реализации для него и участие в нескольких крупномасштабных проектах, базирующихся в значительной степени на технологии обмена сообщениями, на понимание того, что что-то не так с классической клиент/серверной моделью умного сервера обмена сообщениями (брокера) и бессловесными клиентами обмена сообщениями.

Наша главная задача в то время была связана с производительностью: Если в середине находится сервер, то каждое сообщение должно пройти в сети два раза (от отправителя к брокеру и от брокера к приемнику), порождая при этом проблемы, связанные с задержками и пропускной способностью. Более того, если все сообщения передаются через брокера, то в какой-то момент он обязан стать узким местом.

Вторая задача была связана с развертыванием системы в крупномасштабных сетях: когда при развертывании пересекаются организационные границы, то концепция центрального органа управления всем потоком сообщений перестает применяться. Ни одна из компаний не готова уступить контроль сервером другой компании, есть коммерческая тайна и есть юридическая ответственность. Результат на практике состоит в том, что в компании есть один сервер обмена сообщениями с рукописными мостами для подключения к системам обмена сообщениями в других компаниях. Поэтому экосистема в целом сильно фрагментирована, а поддержка большого количества мостов для каждой участвующей компании не делает ситуацию лучше. Чтобы решить эту проблему, нам нужна полностью распределенная архитектура, архитектура, в которой каждым компонентом может управлять, возможно, иной хозяйствующий субъект. Учитывая, что блоком управления в серверной архитектуре является сервер, мы можем решить эту проблему путем установки отдельного сервера для каждого компонента. В таком случае можно дополнительного оптимизировать проект, сделав так, сервер и компонент будут использовать одни и те же процессы. Так, что в итоге у нас мы все это завершилось созданием библиотеки обмена сообщениями.

Проект ØMQ был начат, когда мы получили представление о том, как сделать работу с сообщениями без центрального сервера. Это требует переворот всей концепции сообщений с ног на голову и замены модели автономного централизованного хранилища сообщений в центре сети на архитектуру «умные конечные точки, молчащая сеть», которая базируется на принципе соединений «точка-точка». Техническим следствием этого решения было то, что проект ØMQ с самого начала был библиотекой, а не приложением.

Одновременно мы смогли доказать, что такая архитектура является более эффективной (меньшие задержки, более высокая пропускная способность) и более гибкой (она проще для построения различных сложных топологий, чем привязка к классической модели «ось и спицы»).

Однако, одним из непреднамеренных последствий было то, что выбор в пользу библиотечной модели улучшил удобство работы с продуктом. Снова и снова пользователи выражают свое удовольствие по поводу того, что им не требуется устанавливать автономный сервер обмена сообщениями и им управлять. Оказывается, что отсутствие сервера является предпочтительным вариантом, поскольку это сокращает эксплуатационные затраты (не требуется администратор сервера сообщений) и становится проще выход на рынок (нет необходимости вести переговоры о необходимости запуска сервера с клиентом, командой, осуществляющей управление, или эксплуатационной командой).

Усвоенный урок в том, что при запуске нового проекта вы должны отдавать предпочтение созданию библиотек, если это вообще возможно. Довольно легко создать приложение из библиотеки, запустив ее из тривиальной программы, но практически невозможно создать библиотеку из существующих исполняемых модулей. Библиотека для пользователей будет гораздо более гибкой и в то же время не потребует от них нетривиальных административных усилий.

24.2. Глобальное состояние

Глобальные переменные не всегда хороши при использовании с библиотеками. Библиотека может загружаться в процесс неоднократно, но даже в этом случае будет только один набор глобальных переменных. На рис.24.1 показана библиотека ØMQ, которая используется из двух различных и независимых библиотек. Затем приложение использует обе эти библиотеки.

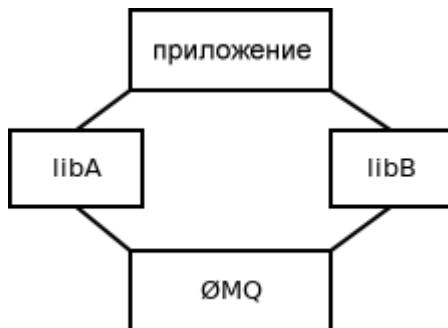


Рис.24.1: ØMQ используется двумя различными библиотеками

Когда возникает такая ситуация, то оба экземпляра ØMQ имеют доступ к одним и тем же переменным, в результате чего возникают состояния гонки, странные сбои и неопределенное поведение.

Чтобы избежать этой проблемы, в библиотеке ØMQ отсутствуют глобальные переменные. Вместо этого пользователь библиотеки ответственен за работу в явном виде с глобальным состоянием. Объект, содержащий глобальные состояния, называется *контекстом*. Хотя с точки зрения пользователя контекст выглядит более или менее похожим на пул рабочих потоков, с точки зрения ØMQ это просто объект для хранения любого глобального состояния, которое нам понадобится. На рисунке, приведенном выше, библиотека `libA` должна иметь свой собственный контекст точно также, как и библиотека `libB`. Тогда ни одна из них не сможет вывести из строя или повлиять на другую библиотеку.

Усвоенный здесь урок довольно очевиден: не используйте в библиотеках глобальное состояние. Если вы это делаете, то в случае, когда в одном и том же процессе будет использовано два экземпляра библиотеки, библиотека выйдет из строя.

24.3. Производительность

Когда проект ØMQ был запущен, его основной целью была оптимизация производительности. Производительность системы обмена сообщениями оценивается с помощью двух метрик: пропускной способности - сколько сообщений может быть передано в течение определенного количества времени, и задержкой - сколько времени требуется сообщению для того, чтобы добраться от одной конечной точки к другой.

На какой показатель мы должны ориентироваться? Какая связь между ними? Разве это не очевидно? Запустите тест, разделите общее время теста на количество пришедших сообщений и вы получаете задержку. Разделите количество сообщений на время и вы получаете пропускную способность. Другими словами, задержка обратно пропорциональна величине пропускной способности. Тривиально, не так ли?

Вместо того, чтобы сразу начать кодирование, мы потратили несколько недель на более подробное исследование метрик производительности и выяснили, что отношения между пропускной способностью и задержкой гораздо более тонкое, чем приведенное выше, и часто метрики довольно нелогичны.

Представьте себе, А отправляет сообщения для В (смотрите рис.24.2.) Общее время теста составляет 6 секунд. Было передано 5 сообщений. Следовательно пропускная способность равна 0,83 сообщения/сек ($5/6$), а задержка равна 1,2 сек ($6/5$), не так ли?

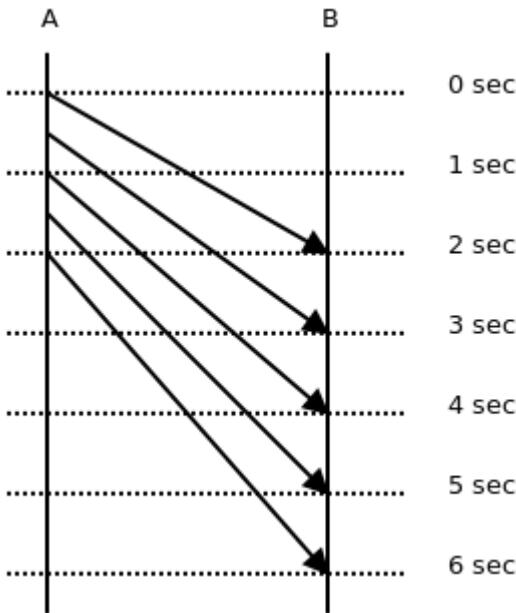


Рис.24.2: Отправка сообщений из А в В

Снова взгляните на диаграмму. На ней видно различное время для каждого сообщения, которое поступает из А в В: 2 сек, 2,5 сек, 3 сек, 3,5 сек, 4 сек. В среднем это 3 секунды, что довольно далеко от нашего первоначального расчета в 1,2 секунды. Этот пример показывает заблуждения, которые интуитивно делаются относительно метрик производительности.

Теперь взгляните на пропускную способность. Общее время теста составляет 6 секунд. Однако, в А будет затрачено всего 2 секунды для того, чтобы отправить все сообщения. С точки зрения А пропускная способность равна 2,5 сообщений/сек ($5/2$). В В затрачивается 4 секунды для того, чтобы принять все сообщения. Так что с точки зрения В пропускная способность равна 1,25 сообщений/сек ($5/4$). Ни одно из этих значений не соответствует нашему первоначальному расчету в 1,2 сообщений/сек.

Короче говоря, задержка и пропускная способность являются двумя различными метриками - это очевидно. Важно понимать различие между ними и их взаимосвязь. Задержку можно измерить только между двумя различными точками в системе; нет такого понятия, как задержка в точке А. Каждое сообщение имеет свою собственную задержку. Вы можете вычислить среднее значение задержек нескольких сообщений, однако, нет такого понятия, как задержка потока сообщений.

Пропускная способность, с другой стороны, может измеряться только в одной точке системы. Имеется пропускная способность отправителя, пропускная способность принимающей стороны, есть пропускная способность любой промежуточной точкой между ними, но нет такого понятия, как общая пропускная способность всей системы. И пропускная способность имеет смысл только для набора сообщений, нет такого понятия, как пропускная способность одного сообщения.

Что касается отношений между пропускной способности и задержкой, то, оказывается, действительно, между ними есть взаимосвязь; однако, в формуле есть интегралы и мы здесь ее обсуждать не будем. Для получения дополнительной информации, читайте литературу по теории очередей.

При оценке производительности системы обмена сообщениями есть еще очень много подводных камней, так что мы не будем вдаваться в подробности. Упор нужно сделать на следующем усвоенном уроке: Убедитесь, что вы понимаете проблему, которую вы решаете. Даже такая проблема, как просто «сделать что-то более быстрым» может для ее правильного понимания потребовать большого объема работы. Более того, если вы не понимаете проблему, вы, вероятно, на основе невидных предположений и популярных мифов создадите код, в результате чего это решение будет

иметь недостатки или, по крайней мере, будет гораздо более сложным или гораздо менее полезным, чем это можно было бы сделать.

24.4. Критический путь

Мы в процессе оптимизации обнаружили, что на производительность оказывают решающее влияние следующие три фактора:

- Количество операций выделения памяти
- Количество системных вызовов
- Модель распараллеливания

Однако не каждая операция выделения памяти и не каждый системный вызов оказывает одинаковое влияние на производительность. Характеристикой, которая нас интересует в системах обмена сообщениями, является количество сообщений, которое мы можем передать между двумя конечными точками в течение определенного количества времени. Кроме того, нас может интересовать то, как долго сообщение передается из одной точки в другую.

Но, учитывая то, что ØMQ предназначена для сценариев с долгоживущими соединениями, время, необходимое для установления соединения или времени, необходимое для обработки ошибки соединения, в основном, значения не имеет. Эти события происходят очень редко, и поэтому их влияние на общую производительность незначительно.

Та часть кода, которая снова и снова используется очень часто, называется *критическим путем* (*critical path*); оптимизировать следует критический путь.

Давайте рассмотрим пример: библиотека ØMQ не очень оптимизирована относительно выделения памяти. Например, при обработке строк, она часто выделяет новую строку для каждого промежуточного этапа преобразования. Тем не менее, если мы посмотрим строго на критический путь, т. е. фактическую передачу сообщений, мы увидим, что в библиотеке практически не происходит выделение памяти. Если сообщения небольшие, то это всего лишь одно выделение памяти на 256 сообщений (эти сообщения хранятся в одном большом выделенном участке памяти). Если, кроме того, поток сообщений устойчив и без огромных пиков трафика, то количество выделений памяти на критическом пути падает до нуля (выделенные участки памяти не возвращаются, но снова и снова используются повторно).

Усвоенный урок: Есть разница в том, где выполнять оптимизацию. Оптимизация фрагментов кода, которые не находятся на критическом пути, является напрасной тратой усилий.

24.5. Выделение памяти

Если предположить, что вся инфраструктура инициализирована и соединение между двумя конечными точками уже установлено, есть только одно, для чего выделяется память, это - само сообщение. Таким образом, для оптимизации критического пути мы должны были изучить, как происходит выделение памяти под сообщения и как сообщения передаются вверх и вниз по стеку.

В сфере высокопроизводительных сетей общеизвестно, что лучшая производительность достигается за счет четкого баланса между стоимостью выделения памяти под сообщение и стоимостью копирования сообщения (например, <http://hal.inria.fr/docs/00/29/28/31/PDF/Open-MX-IOAT.pdf>: сравните различные подходы для «малых», «средних» и «больших» сообщений). Для небольших сообщений, копирование гораздо дешевле, чем выделение памяти. Имеет смысл вообще не выделять никаких новых кусков памяти, а вместо этого по мере необходимости копировать сообщение в заранее выделенную память. Для больших сообщений, с другой стороны, копирование гораздо

дороже, чем выделение памяти. Имеет смысл один раз выделить память для сообщения и, вместо копирования данных, передать указатель на выделенный блок. Такой подход называется «нулевым копированием».

ØMQ обрабатывает оба варианта прозрачно. Сообщение ØMQ представлено структурой, в которой спрятаны детали. Содержимое очень маленьких сообщений хранится непосредственно в структуре. Таким образом, при создании копии структуры действительно копируется данные сообщения. Когда сообщение станет большим, для него выделяется отдельный буфер, а в структуре хранится только указатель на буфер. Создание копии структуры не приводит к копированию данных сообщения, что разумно в случае, когда размер сообщения равен мегабайтам (рис. 24.3). Следует отметить, что в последнем случае в буфере подсчитывается количество указателей, так что можно использовать указатели из нескольких структур и не требуется копировать данные.

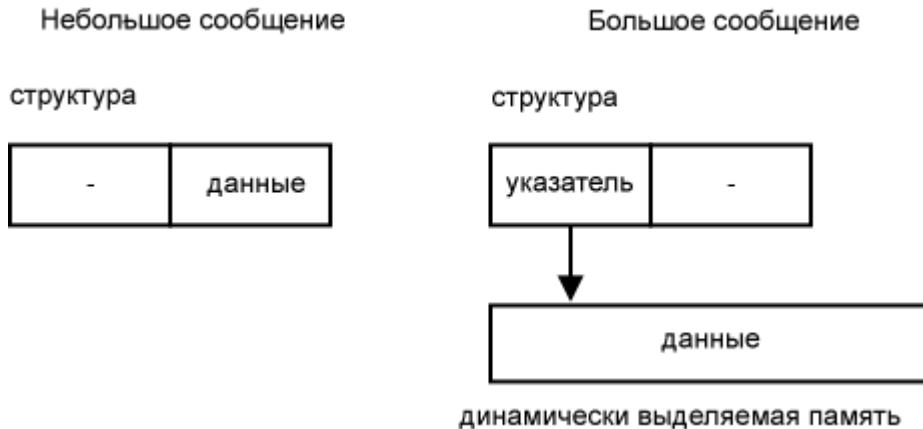


Рис.24.3: Копирование сообщения (или только указателя)

Усвоенный урок: Когда вы думаете о производительности, то не считайте, что наилучшее решение только одно. Может случиться, что есть несколько подклассов проблемы (например, небольшие сообщения и большие сообщения), для каждого из которых есть свой собственный оптимальный алгоритм.

24.6. Пакетная обработка

Как уже было упомянуто, огромное количество системных вызовов в системе обмена сообщениями может привести к возникновению узких мест по производительности. На самом деле, проблема гораздо более общая. Возникают очень нетривиальные потери производительности, связанные с обходом стека вызовов и, следовательно, при создании высокопроизводительных приложений разумно настолько, насколько это возможно, избегать выполнение обхода стека вызовов.

Рассмотрим рис.24.4. Чтобы отправить четыре сообщения, вы должны четыре раза пройти весь сетевой стек целиком (т.е. ØMQ, glibc, границу пользовательского пространства/пространства ядра, реализацию TCP, реализацию IP, слой Ethernet, сам NIC и снова вернуться).

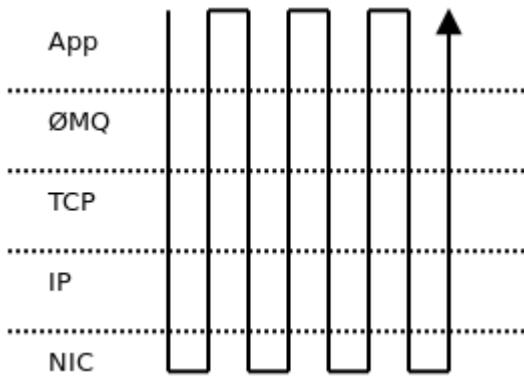


Рис.24.4: Отправка четырех сообщений

Тем не менее, если вы решите объединить эти четыре сообщения в один пакет, то потребуется только один обход стека (рис. 24.5). Влияние на пропускную способность сообщений может быть огромным: до двух порядков, особенно если сообщения маленькие и сотни таких сообщений можно упаковывать в один пакет.

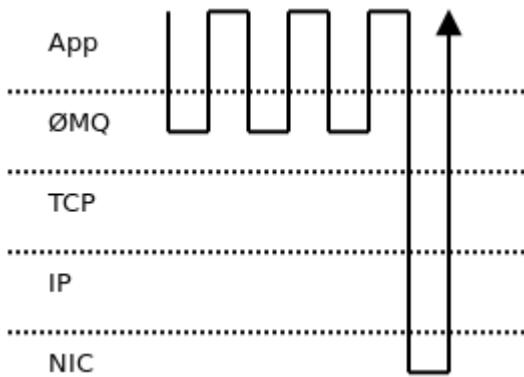


Рис.24.5: Пакетная обработка сообщений

С другой стороны, пакетная обработка может иметь негативное влияние на задержку. Возьмем, например, хорошо известный алгоритм Нэйгла (Nagle), который реализован в TCP. Он задерживает исходящие сообщения в течение определенного количества времени и объединяет все накопленные данные в одном пакете. Очевидно, что полная задержка первого сообщения в пакете гораздо больше, чем задержка последнего. Поэтому обычно, чтобы в приложениях снизить задержку, алгоритм Нэйгла отключается. Обычно отключается даже пакетная обработка на всех уровнях стека (например, возможность объединения прерываний NIC).

Но опять же, отсутствие пакетной обработки не означает больших перемещений по стеку и не приводит к низкой пропускной способности сообщений. Мы, кажется, столкнулись с дилеммой между пропускной способностью и задержкой.

Библиотека ØMQ пытается обеспечить сравнительно низкие задержки в сочетании с высокой пропускной способностью за счет использования следующей стратегии: когда поток сообщений не большой и не превышает пропускную способность сетевого стека, ØMQ отключает всю пакетную обработку с тем, чтобы улучшить задержку. Компромисс здесь в несколько большем использовании ЦП - мы все еще должны часто проходить через стек. Однако в большинстве случаев это не является проблемой.

Когда скорость сообщений превышает пропускную способность сетевого стека, сообщения должны быть поставлены в очередь и храниться в памяти до тех пор, пока стек не будет готов принять их. Очередь означает, что задержка будет расти. Если сообщение находится в очереди одну секунду, то полная задержка будет равна, по меньшей мере, одну секунду. Что еще хуже, поскольку

размер очереди растет, задержка будет постепенно увеличиваться. Если размер очереди не ограничен, то задержка может быть больше любого заранее заданного предела.

Было обнаружено, что даже если сетевой стек настроен на минимально возможную задержку (выключен алгоритм Нэйгла, выключено объединение прерываний NIC и т.д.) задержка все еще может быть достаточно большой из-за эффекта очереди, описанного выше.

В такой ситуации имеет смысл агрессивно начинать использование пакетной обработки. Нет ничего, чтобы можно было потерять, поскольку в любом случае задержка и так уже высока. С другой стороны, агрессивные использование пакетной обработки увеличивает пропускную способность и может убрать из очереди ожидающие сообщения, что в свою очередь означает, что задержка будет постепенно падать поскольку задержка из-за очереди уменьшается. Как только в очереди станет мало сообщений, пакетная обработка может быть отключена для еще большего снижения задержки.

Еще одно наблюдение состоит в том, что пакетная обработка должна выполняться только на самом верхнем уровне. Если сообщения группируются там, нижние слои так или иначе не имеют никакого отношения к пакетной обработке, поскольку все алгоритмы пакетной обработки ничего не делают, кроме как вводят дополнительную задержку.

Усвоенный урок: Для того, чтобы в асинхронной системе получить оптимальную пропускную способность в сочетании с оптимальным временем ответа, выключите все алгоритмы пакетной обработки на низких уровнях стека и включите пакетную обработку на самом верхнем уровне. Пакетная обработка требуется только тогда, когда новые данные прибывают быстрее, чем они могут быть обработаны.

24.7. Общий обзор архитектуры

До этого момента мы сосредоточили внимание на общих принципах, которые делают библиотеку ØMQ быстрой. Теперь мы взглянем на реальную архитектуру системы (рис. 24.6).

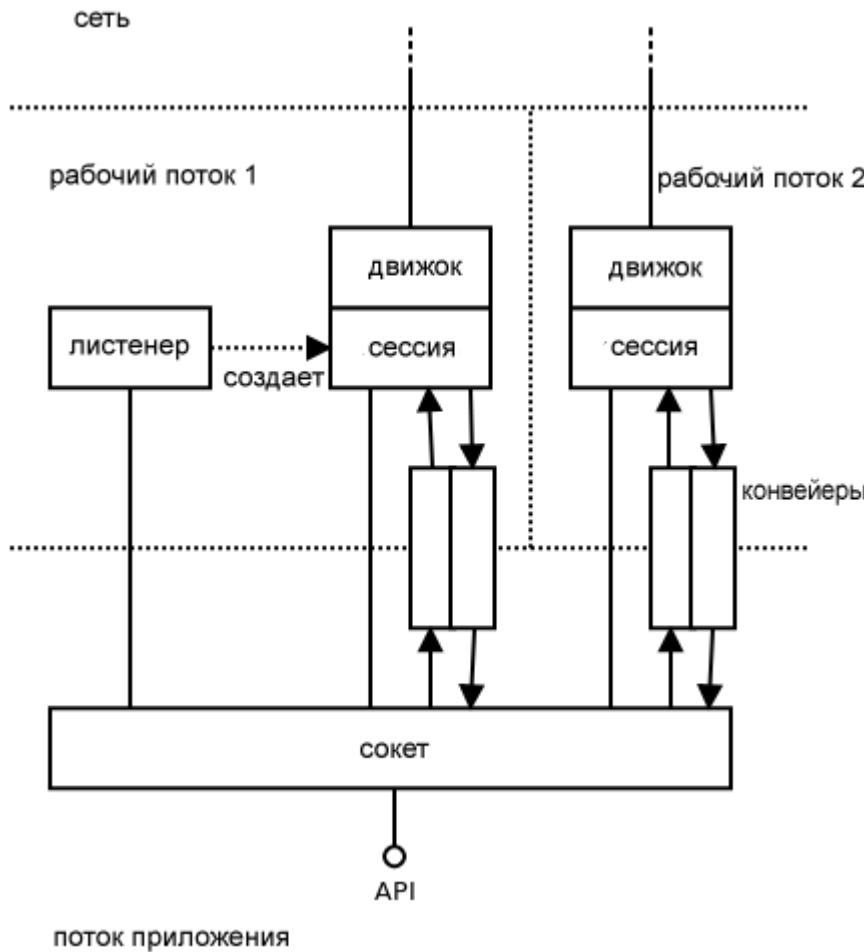


Рис.24.6: Архитектура ØMQ

Пользователь взаимодействует с ØMQ с использованием так называемых «сокетов». Они очень похожи на сокеты TCP, основное отличие в том, что каждый сокет может обрабатывать соединения с несколькими абонентами, что немного похоже на то, как это делают несвязанные сокеты UDP.

Объект сокета живет в потоке пользователя (смотрите обсуждение потоковых моделей в следующем разделе). Кроме этого, ØMQ работает в нескольких рабочих потоках, которые обрабатывают асинхронную часть соединения: чтение данных из сети, помещение сообщений в очередь, прием поступающих соединений и т.д.

Существуют различные объекты, находящиеся в рабочих потоках. Каждый из этих объектов принадлежит точно только одному родительскому объекту (принадлежность на диаграмме обозначается простой сплошной линией). Родительский объект может находиться в потоке, отличном от потока потомка. Большинство объектов принадлежат непосредственно сокетам; однако, есть несколько случаев, когда объект находится в собственности объекта, который принадлежит сокету. Все, что мы получаем, это дерево объектов, причем по одному такому дереву на сокет. Дерево используется в ходе завершения работы с сокетами; работа ни с одним из объектов не может быть завершена прежде, чем будет завершена работа со всеми его потомками. Таким образом, мы можем гарантировать, что процесс завершения будет работать так, как ожидается, например, ожидающие исходящие сообщения будут отправлены в сеть до завершения процесса отправки.

Грубо говоря, есть два вида асинхронных объектов; есть объекты, которые не участвуют в передаче сообщений, и есть объекты, которые участвуют. Объекты первого вида связаны главным образом с управлением соединением. Например, объект слушателя TCP (листенер) прослушивает входящие соединения TCP и создает объекты движка/сеанса для каждого нового соединения. Анало-

гичным образом объект коннектора TCP (соединение) пытается подключиться к порту TCP и, в случае успеха, создает объект движка/сессии для управления подключением. Если соединение было разорвано, то объект коннектора пытается восстановить его (реконнектор).

Объекты второго вида представляют собой объекты, которые непосредственно участвуют в передаче данных. Эти объекты состоят из двух частей: *сессионный объект* (*session object*) отвечает за взаимодействие с сокетом ØMQ, а *объект движка* (*engine object*) отвечает за связь с сетью. Имеется только один вид сессионного объекта, но для каждого протокола, который поддерживается в ØMQ, имеются различные типы объектов движков. Т.е., у нас есть движки TCP, движки IPC (межпроцессное взаимодействие), движки PGM (надежный мультикастовый протокол —смотрите RFC 3208) и др. Набор движков можно расширять - в будущем мы можем выбрать для реализации, скажем, движок WebSocket или движок SCTP.

Сессии являются сеансами обмена сообщениями с сокетами. Есть два направления для передачи сообщений и каждое направление обрабатывается при помощи конвейерного объекта. Каждый конвейер является в своей основе очередью без блокировок, оптимизированной для быстрого прохождения сообщений между потоками.

Наконец, есть объект контекста (о нем рассказывалось в предыдущих разделах, но он не показан на рисунке), в котором хранится глобальное состояние и он доступен всем сокетам и всем асинхронным объектам.

24.8. Модель распараллеливания

Одним из требований для ØMQ было возможность использования многоядерных устройств; другими словами, чтобы можно было масштабировать пропускную линейно с увеличением числа доступных ядер процессора.

Наш предыдущий опыт работы с системами обмена сообщениями показал, что с использованием нескольких потоков в классическом пути (критические секции, семафоры и т.д.) не дает значительного улучшения производительности. В самом деле, многопоточная версия системы обмена сообщениями может быть более медленной, чем однопоточная, даже если измеренная осуществляется на многоядерном устройстве. Отдельные потоки просто тратят слишком много времени на ожидание друг друга, и в то же время требуют большого количества переключений контекста, что замедляет работу системы.

Учитывая эти проблемы, мы решили перейти на другую модель. Цель состояла в том, чтобы полностью избежать блокировок и позволить каждому потоку работать на полной скорости. Взаимодействие между потоками было реализовано с помощью асинхронных сообщений (события), которые передаются между потоками. Это, как знают инсайдеры, является классической *моделью актера* (*actor model*).

Идея заключалась в том, чтобы запускать на каждом ядре процессора по одному рабочему потоку — наличие двух потоков, совместно использующих то же самое ядро, будет означать лишь большое количество переключений контекста без получения особых преимуществ. Каждый внутренний объект ØMQ, такой, как, скажем, движок TCP, будет тесно связан с конкретным рабочим потоком. Это, в свою очередь, означает, что нет никакой необходимости в критических секциях, взаимоисключаемых событиях (mutexes), семафорах и тому подобном. Кроме того, эти объекты ØMQ не будут перераспределяться между ядрами процессора, так что удастся избежать негативного влияния на производительность, связанного с загрязнением кэша (рис.24.7).

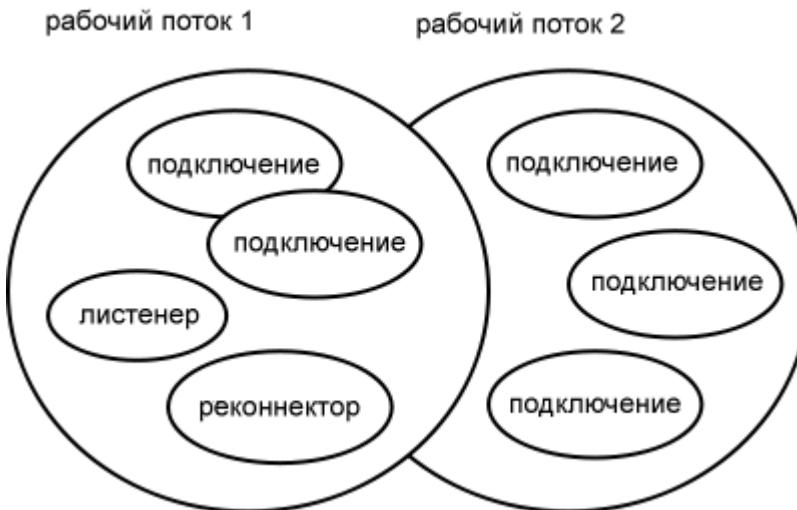


Рис.24.7: Несколько рабочих потоков

Благодаря такой конструкции исчезает много традиционных многопоточных проблем. Тем не менее, есть необходимость в том, чтобы рабочим потоком могли пользоваться множество объектов, что в свою очередь означает, что должен быть какой-то вид кооперативной многозадачности. Это означает, что нам нужен планировщик; объекты должны управляться при помощи событий, не надо реализовывать управление циклом всех событий; мы должны учесть возможность возникновения событий в произвольной последовательности, причем даже очень редких; мы должны обеспечить, чтобы ни один объект не удерживал процессор слишком долго и т.д.

Короче говоря, вся система должна стать полностью асинхронной. Никакой объект не должен выполнять блокирующих операций, поскольку он заблокирует не только себя, но также и все другие объекты, использующие тот же самый рабочий поток. Все объекты должны представлять собой, прямо или косвенно, автоматы или машины состояний. При наличии сотен или тысяч машин состояний, работающих параллельно, вы должны обеспечить все возможные взаимодействия между ними и, самое главное, правильно реализовать процесс завершения их работы.

Получается, что завершение работы полностью асинхронной системы является в чистом виде устраивающе сложной задачей. При попытке завершить работу тысячи движущихся частей, некоторые из которых работают, некоторые находятся в состоянии ожидания, некоторые - в процессе инициализации, некоторые из них уже завершили свою работу самостоятельно, возможны возникновения всех видов состояний гонки, утечки ресурсов и тому подобное. Подсистема завершения работы является, безусловно, самой сложной частью ØMQ. Быстрый просмотр трекера ошибок показывает, что около 30 - 50% обнаруженных ошибок связаны в той или иной форме с этапом завершения работы системы.

Усвоенный урок: Когда стремитесь к экстремальной производительности и масштабируемости, то рассмотрите модель актера; это чуть ли не единственная вариант в подобных случаях. Однако, если вы не пользуетесь специализированной системой, например, Erlang или самой ØMQ, вам придется написать и вручную отладить инфраструктуру большого объема. Кроме того, с самого начала подумайте о процедуре завершения работы системы. Это будет самая сложная часть кода, и если у вас нет четкого представления о том, как ее реализовать, вам, вероятно, следует в первую очередь пересмотреть использование модели актера.

24.9. Неблокирующие алгоритмы

В последнее время в моде стали неблокирующие алгоритмы. Это простые механизмы межпотокового взаимодействия, в которых не используются предоставляемые ядром примитивы синхронизации, такие как взаимоисключающие события или семафоры; они предпочитают выполнять син-

хронизацию с использованием атомарных операций процессора, таких как атомное сравнение и своп (CAS). Следует иметь в виду, что они не в буквальном смысле работают без блокировок — в действительности блокировки происходят за кулисами на аппаратном уровне.

ØMQ использует неблокирующую очередь в конвейерных объектах для передачи сообщений между потоками пользователя и рабочими потоками ØMQ. Есть два интересных аспекта, касающихся того, как ØMQ использует неблокирующую очередь.

Во-первых, в каждой очереди есть ровно один поток, который осуществляется запись, и ровно один поток, который осуществляет чтение. Если необходима связь типа «1-N», то создается несколько очередей (рис.24.8). Благодаря такой организации очереди не надо беспокоиться о синхронизации записи (есть только один поток, осуществляющий запись) или чтения (есть только один поток, осуществляющий чтение), причем очередь может быть реализована очень эффективным способом.



Рис.24.8: Очереди

Во-вторых, мы поняли, что хотя неблокирующие алгоритмы более эффективны, чем классические алгоритмы, базирующиеся на использовании взаимоисключаемых событий, атомарные операции процессора все еще достаточно дороги (особенно когда есть рассогласованность между ядрами процесса) и выполнение атомарной операции для каждого сообщения, которое записывается или читается, происходит гораздо медленнее, чем нам это могло подойти.

Способ ускорения — опять же - потоковая обработка. Представьте, что у вас есть 10 сообщений, которые должны быть записаны в очередь. Это может произойти, например, когда вы получили сетевой пакет, содержащий 10 небольших сообщений. Получение пакета является атомарным событием, вы не можете получить половину пакета. В результате этого атомарного события в неблокирующую очередь потребуется записать 10 сообщений. Нет никакого смысла выполнять атомарную операцию для каждого сообщения. Вместо этого, вы можете накопить сообщения в виде порции «предзаписи» в той части очереди, которая доступна исключительно для записывающего потока, а затем сбросить ее в очередь с помощью одной атомарной операции.

То же самое относится и к считыванию из очереди. Представьте себе 10 сообщений, рассмотренных выше, которые уже были помещены в очередь. Поток, осуществляющий чтение, может извлекать каждое сообщение из очереди с использованием атомарной операции. Тем не менее, это перебор; вместо этого, поток может с помощью одной атомарной операции перенести все ожидающие сообщения в порцию «предварительного чтения» в очереди. После этого, он может выбирать сообщения из буфера «предварительного чтения» по одному. Порция «предварительного чтения» принадлежит и доступна исключительно потоку, осуществляющему чтение, и, таким образом, на этой фазе вообще не нужна какая-либо синхронизация.

Стрелка в левой части рисунка 24.9 показывает, как можно с помощью модификации одного указателя выполнить сброс в очередь содержимого буфера предварительной записи. Стрелка в правой

части показывает, как можно ничего не делая, а только изменив еще один указатель, сдвинуть содержимое очереди в буфер предварительного чтения.

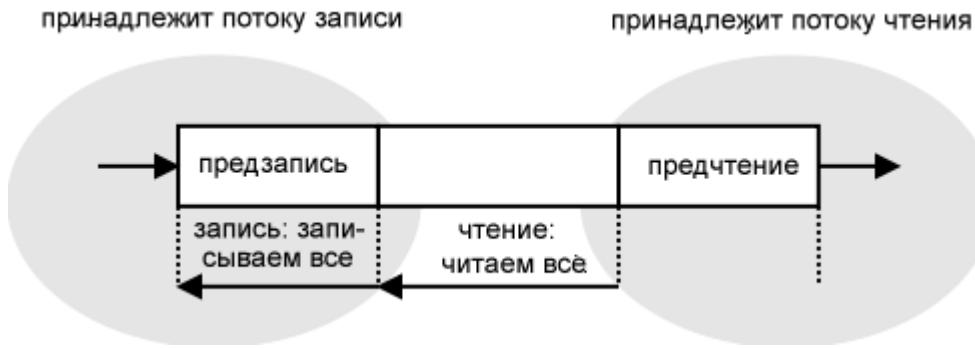


Рис.24.9: Неблокирующая очередь

Усвоенный урок: неблокирующие алгоритмы трудно придумывать, сложно реализовывать и почти невозможно отлаживать. Если возможно, то используйте существующие проверенные алгоритмы, а не изобретайте свои собственные. Если требуется экстремальная производительность, то не следует полагаться исключительно на неблокирующие алгоритмы. Хотя они и быстрые, производительность можно значительно улучшить, если поверх их сделать умную пакетную обработку.

24.10. Интерфейс API

Пользовательский интерфейс является наиболее важной частью любого продукта. Это единственная часть вашей программы, которая видна внешнему миру, и если вы сделаете ее неправильно, то мир будет ненавидеть вас. В конечном изделии это либо графический интерфейс, либо интерфейс командной строки. В библиотеках - это интерфейс API.

В ранних версиях ØMQ интерфейс API базировался на модели обменов и очередей AMQP (смотрите [спецификации AMQP](#)). С исторической точки зрения было бы интересно взглянуть на документ [white paper from 2007](#), в котором делается попытка примирить AMQP с бесброкерной моделью обмена сообщениями. Я потратил окончание 2009 года на его переписывание почти с нуля, чтобы вместо этого использовать интерфейс BSD Socket API. Это был поворотный момент; с этого момента количество применений библиотеки ØMQ взлетело. Если раньше это был нишевый продукт, используемый только горсткой экспертов по сообщениям, то после этого он стал обычным инструментом, удобным для любого. Через год или около того размер сообщества увеличился в десять раз, было реализовано несколько привязок к 20 различным языкам и т.д.

Пользовательский интерфейс определяет восприятие продукта. При практически оставшихся тех же самых функциональных возможностях - просто за счет изменения интерфейса API - ØMQ превратился из продукта уровня «enterprise messaging» (промышленной системы обмена сообщениями) в «сетевой» продукт. Иными словами, восприятие изменилось со «сложной части инфраструктуры для крупных банков» на то, что «это помогает мне отправить мое сообщение длиной в 10 байт из приложения А в приложение В».

Усвоенный урок: Разберитесь с тем, каким, как вы хотите, должен быть ваш проект, и проектируйте соответствующий пользовательский интерфейс. Если у вас есть пользовательский интерфейс, который не совпадает с видением проекта, то это является 100% гарантией движения к провалу.

Одним из важных аспектов перехода на интерфейс BSD Sockets API было то, что он не был революционным недавно изобретенным API, а уже существовал и был хорошо известен. На самом деле, интерфейс BSD Sockets API является одним из старейших API, которым сегодня по-прежнему

активно пользуются; он восходит к 1983 году и к 4.2BSD Unix. Он широко распространен и стабилен в течение буквально десятилетий.

Вышеуказанный факт дает много преимуществ. Во-первых, это интерфейс API, который известен каждому, поэтому кривая обучения будет до неприличия плоской. Даже если вы никогда не слышали о ØMQ, вы можете создать свое первое приложение через пару минут благодаря тому, что вы можете воспользоваться знаниями о BSD Sockets.

Во-вторых, использование широко распространенного интерфейса API позволяет интегрировать ØMQ с существующими технологиями. Например, представление объектов ØMQ в виде «сокетов» или «дескрипторов файлов» позволяет в одном и том же цикле событий выполнять обработку событий TCP, UDP, конвейеров, файлов и ØMQ. Другой пример: [экспериментальный проект](#) по переносу функций, похожих на ØMQ, в ядро Linux оказывается в реализации довольно простым. За счет применения того же самого по концептуальности фреймворка уже сейчас можно пользоваться большей частью инфраструктуры ØMQ.

Третьим и, вероятно, самым главным, является тот факт, что поскольку интерфейс BSD Sockets API выжил в течение почти трех десятилетий несмотря на многочисленные попытки его заменить, это означает, что в нем самом изначально есть нечто. Разработчики BSD Sockets API, намеренно или случайно, приняли правильные проектные решения. Используя это API, мы можем автоматически воспользоваться этими проектными решениями даже не зная, какими они были и какие проблемы они решали.

Усвоенный урок: Хотя интерес к повторному использованию кода был повышен с незапамятных времен, а позже к нему присоединилось повторное использование шаблонов, важно думать о повторном использовании решений в еще более общем виде. Когда разрабатываете продукт, взгляните на аналогичные продукты. Проверьте, какие не удались, а какие оказались успешными; изучите успешные проекты. Не поддавайтесь синдрому «Здесь ничего не придумано». Повторно используйте идеи, интерфейсы API, концептуальные фреймворки, которые вы посчитаете подходящими. Поступая таким образом, вы позволяете пользователям повторно использовать имеющиеся у них знания. В то же время вы можете избежать технических ловушек даже в случае, если вы на данный момент их не осознаете.

24.11. Шаблоны обмена сообщениями

В любой системе обмена сообщениями, наиболее важной проблемой проекта является то, каким способом пользователю приходится указывать какие сообщения направляются и в каком направлении. Существуют два основных подхода, и я считаю, что такая дилемма вполне универсальна и применима практически для любой проблемы, возникающей в области программного обеспечения.

Один из подходов заключается в принятии философии Unix «делать одно и делать это хорошо». Это означает, что область проблемы должна быть искусственно ограничена небольшой и хорошо понятной частью. Затем программа должна решить эту ограниченную задачу правильно и исчерпывающе. Примером такого подхода в сфере обмена сообщениями является [MQTT](#). Это протокол распределенных сообщений для группы потребителей. Он не может использоваться ни для чего другого (скажем, для RPC), но он прост в использовании и хорошо работает с распределенными сообщениями.

В другом подходе мы ориентируемся на общность и предоставляем мощную и глубоко настраиваемую систему. Примером такой системы является AMQP. Его модель очередей и обменов сообщениями предоставляет пользователям средства для программного определения практически любого алгоритма маршрутизации, который они могут придумать. Компромиссом, конечно, является множество параметров, о которых нужно позаботиться.

В ØMQ выбрана первая модель, поскольку в ней предполагается, что полученным продуктом сможет пользоваться в основном каждый, тогда как обобщенная модель для того, чтобы ей пользоваться, требует экспертов в области обмена сообщений. Чтобы это продемонстрировать, давайте посмотрим, как модель влияет на сложность API. Далее приведена реализация клиентской части RPC, построенная поверх обобщенной системы (AMQP):

```
connect ("192.168.0.111")
exchange.declare (exchange="requests", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
exchange.declare (exchange="replies", type="direct", passive=false,
    durable=true, no-wait=true, arguments={})
reply-queue = queue.declare (queue="", passive=false, durable=false,
    exclusive=true, auto-delete=true, no-wait=false, arguments={})
queue.bind (queue=reply-queue, exchange="replies",
    routing-key=reply-queue)
queue.consume (queue=reply-queue, consumer-tag="", no-local=false,
    no-ack=false, exclusive=true, no-wait=true, arguments={})
request = new-message ("Hello World!")
request.reply-to = reply-queue
request.correlation-id = generate-unique-id ()
basic.publish (exchange="requests", routing-key="my-service",
    mandatory=true, immediate=false)
reply = get-message ()
```

С другой стороны, ØMQ разбивает ландшафт обмена сообщениями на так называемые «шаблоны сообщений». Примерами шаблонов являются «публикация/подписка», «запрос/ответ» или «распараллеливаемый конвейер». Каждый шаблон обмена сообщениями полностью ортогонален другим шаблонам и может рассматриваться как отдельный инструмент.

Далее приведена еще одна реализация вышеупомянутого приложения, выполненная в ØMQ с использованием шаблона «запрос/ответ». Обратите внимание на то, что все настройки сводятся к одному шагу выбора правильного шаблона обмена сообщениями («REQ»):

```
s = socket (REQ)
s.connect ("tcp://192.168.0.111:5555")
s.send ("Hello World!")
reply = s.recv ()
```

До этого момента мы утверждали, что конкретные решения лучше, чем обобщенные. Мы хотим, чтобы наши решения, чтобы быть как можно более конкретными. Тем не менее, мы одновременно хотим, насколько это окажется возможным, предложить нашим клиентам максимально широкий спектр функциональных возможностей. Как мы можем решить это кажущееся противоречие?

Ответ состоит из двух шагов:

1. Определение слоя стека, который касается конкретной проблемной области (например, транспорт, маршрутизация, презентация и т.д.).
2. Предоставление нескольких реализаций слоя. Это должны быть непересекающиеся реализации, отдельные для каждого варианта использования.

Давайте посмотрим на пример транспортного уровня стека Internet. Он предназначен для предоставления таких сервисов, как передача потоков данных, управления потоками, обеспечение надежности передачи и т.д., которые реализуются поверх сетевого уровня (IP). Он реализуется в соответствие с определением нескольких непересекающихся решений: TCP для соединений, ориентированных на надежную передачу потока, UDP для соединения с ненадежной пакетной передачей, SCTP для передачи нескольких потоков, DCCP для ненадежных соединений и так далее.

Обратите внимание, что каждая реализация полностью ортогональна: конечная точка UDP не может обмениваться сообщениями с конечной точкой TCP. А конечная точка SCTP не может обмениваться сообщениями с конечной точкой DCCP. Это означает, что в любой момент к стеку могут быть добавлены новые реализации без влияния на существующие части стека. И наоборот, о неудачных реализациях можно будет забыть и можно будет выбросить их без ущерба для жизнеспособности транспортного уровня в целом.

Тот же принцип относится и к шаблонам обмена сообщениями так, как это сделано в ØMQ. Шаблоны обмена сообщениями формируют слой (так называемый «слой масштабируемости») поверх транспортного уровня (TCP и аналоги). Реализациями этого слоя являются индивидуальные шаблоны обмена сообщениями. Они строго ортогональны — конечная точка «публикации/подписки» не может обмениваться сообщениями с конечной точкой «запроса/ответа» и т.д. Строгое разделение между шаблонами в свою очередь означает, что по мере необходимости могут быть добавлены новые шаблоны и что неудачные эксперименты с новыми шаблонами не повредят существующим шаблонам.

Усвоенный урок: Когда решается сложная и многогранная проблема, то может оказаться, что монолитное обобщенное решение может оказаться не лучшим. Вместо этого, мы можем рассмотреть проблемную область в виде абстрактного слоя и предложить несколько его реализаций, каждая из которых направлена на вполне конкретные условия использования. Когда вы так поступаете, то очень тщательно определите условия использования. Проверьте, что попадает в эту область, а что - нет. Из-за слишком агрессивного ограничения области использования, применение программного обеспечения может быть ограничено. Но если вы определите проблему слишком широко, продукт может стать слишком сложным, нечетким и запутанным для пользователей.

24.12. Заключение

Поскольку наш мир заселяется большим количеством маленьких компьютеров, подключаемых через Интернет - мобильными телефонами, считывателями меток RFID, планшетами и ноутбуками, устройствами GPS и т. д. - проблема распределенных вычислений перестает быть областью академической науки и становится повседневной проблемой, которую решает каждый разработчик. Решения, к сожалению, в основном являются предметно-ориентированными хакерскими трюками. Эта статья подытоживает наш опыт построения крупномасштабной распределенной системы на систематической основе. Она фокусируется на проблемах, которые представляют интерес с точки зрения архитектуры программ, и мы надеемся, что разработчики и программисты из сообщества сторонников открытого кода посчитают ее полезной.