



САЛД
САНКТ-ПЕТЕРБУРГСКАЯ
АНТИВИРУСНАЯ
ЛАБОРАТОРИЯ
ДАНИЛОВА

www.SALDrus
8 (812) 336-3739

Антивирусные
программные продукты

Я неоднократно встречал людей, сокрушающихся о том, что никто не учит студентов и молодых программистов видеть красоту в программировании. Архитекторы изучают постройки других архитекторов, композиторы — произведения, написанные их коллегами, программисты же обращаются к чужим программам лишь с одной целью — найти ошибки и исправить «баги».

Эта книга является попыткой изменить сложившуюся ситуацию. Однажды я попросил некоторых известных разработчиков программного обеспечения изучить и обсудить наиболее совершенные части программного кода, когда-либо попадавшиеся им на глаза. И, как вы сможете убедиться, прочитав эту книгу, они нашли красоту и совершенство во многих аспектах программирования.

Грег Уилсон, составитель книги, редактор журнала *Dr. Dobb's Journal*

Все авторские гонорары от продаж этой книги будут пожертвованы в пользу организации «Международная амнистия» (Amnesty International).

Тема: Программирование. Основы и алгоритмы

Уровень пользователя: эксперт

O'REILLY®

ISBN: 978-5-4237-0331-8



ПИТЕР®

Заказ книг:

197198, Санкт-Петербург, алл. 127
тел.: (812) 759-41-45, 751-10-02; piter@shanson.piter.com
81093, Харьков-83, ал. 9130
тел.: (057) 759-41-45, 751-10-02; piter@shanson.piter.com

www.piter.com — вся информация о книгах и веб-магазин

ИДЕАЛЬНЫЙ КОД

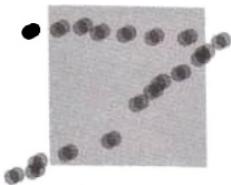
Как научиться видеть красоту в программировании

Под редакцией
Энди Орама и Грега Уилсона

O'REILLY®



Beautiful Code

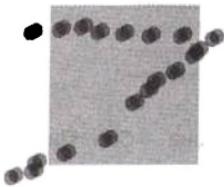


Edited by Andy Oram and Greg Wilson

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

ИДЕАЛЬНЫЙ КОД



Под редакцией Энди Орама и Грэга Уилсона



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2011

Под редакцией Энди Орама и Грэга Уилсона

Идеальный код

Переведен с английского Н. Вильчинский, А. Смирнов

Заведующий редакцией	А. Кривцов
Руководитель проекта	А. Юрческо
Ведущий редактор	О. Некруткина
Литературные редакторы	О. Некруткина, Н. Царенская
Художественный редактор	А. Адуевская
Корректоры	В. Нечаева, Н. Солицева
Верстка	Л. Харитонов

О-63 Идеальный код / Под редакцией Э. Орама и Г. Уилсона — СПб.: Питер, 2011. — 624 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4237-0331-8

В этой уникальной книге самые авторитетные разработчики программного обеспечения делятся опытом оригинального решения задач, которые вставали перед ними при реализации крупных IT-проектов.

С помощью этого издания читатель получит возможность оказаться на месте ведущих программистов, увидеть собственными глазами проблемы, возникавшие при реализации разнообразных проектов, и пройти увлекательный путь их преодоления. Авторские статьи отобраны и отредактированы Грэгом Уилсоном, редактором журнала «Dr. Dobb's Journal», одного из самых авторитетных IT-изданий в мире, а также редактором издательства O'Reilly Энди Орамом.

Одни лишь только список авторов делает эту книгу настоящим бестселлером — здесь вы найдете материалы, написанные такими признанными профессионалами, как Чарльз Петцольд, Джон Бентли, Тим Брай, Брайан Керниган, и еще тридцать четырьмя экспертами в области разработки программного обеспечения.

© O'Reilly

© Перевод на русский язык ООО Издательство «Питер», 2011

© Издание на русском языке, оформление ООО Издательство «Питер», 2011

Права на издание получены по соглашению с O'Reilly.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-4237-0331-8

ISBN 9780596510046 (англ.)

Подписано в печать 08.12.10. Формат 70x100/16. Усл. л. л. 50,31. Тираж 1000. Заказ 24781.

ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2: 95 3005 · литература учебная.

Отпечатано по технологии СПР в ОАО «Печатный двор» им. А. М. Горького.

197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Введение	16
Глава 1. Поиск соответствий с помощью регулярных выражений	17
<i>Брайен Керниган</i>	
Глава 2. Дельта-редактор Subversion: Интерфейс и онтология	27
<i>Карл Фогель</i>	
Глава 3. Самый красивый код, который я никогда не писал	46
<i>Джон Бентли</i>	
Глава 4. Поиск вещей	60
<i>Тим Брэй</i>	
Глава 5. Правильный, красивый, быстрый (именно в таком порядке): уроки разработки XML-верификаторов	79
<i>Элиот Расти Гарольд</i>	
Глава 6. Платформа для проведения комплексного теста: красота, полученная за счет хрупкости конструкции.	96
<i>Майкл Фезерс</i>	
Глава 7. Красивые тесты	107
<i>Альберто Савойя</i>	
Глава 8. Динамическая генерация кода для обработки изображений	128
<i>Чарльз Петцольд</i>	
Глава 9. Нисходящая иерархия операторов	152
<i>Дуглас Крокфорд</i>	

Глава 10. Поиск методов ускоренного подсчета заполнения	170
<i>Генри Уоррен-мл.</i>	
Глава 11. Безопасная связь: технология свободы	186
<i>Ашиш Гулхати</i>	
Глава 12. Становление красивого кода в BioPerl	217
<i>Линкольн Стейн</i>	
Глава 13. Конструкция генного сортировщика	249
<i>Джим Кент</i>	
Глава 14. Как первоклассный код развивается вместе с аппаратным обеспечением (на примере Гауссова исключения)	263
<i>Джек Донгарра и Петр Лушчек</i>	
Глава 15. Долговременные выгоды от красивой конструкции	290
<i>Адам Колава</i>	
Глава 16. Модель драйверов ядра Linux: преимущества совместной работы	305
<i>Грег Кроа-Хартман</i>	
Глава 17. Иной уровень косвенного подхода	317
<i>Диомидис Спинеллис</i>	
Глава 18. Реализация словарей Python: стремление быть всем во всем полезным	331
<i>Эндрю Кучлинг</i>	
Глава 19. Многомерные итераторы в NumPy	341
<i>Трейвис Олифант</i>	
Глава 20. Высоконадежная корпоративная система, разработанная для миссии NASA Mars Rover.	359
<i>Рональд Мак</i>	
Глава 21. ERP5: Конструирование с целью достижения максимальной адаптивности	381
<i>Роджерио Эйтем де Карвальо и Рафаэль Моннерат</i>	

Глава 22. Ложка грязи	396
<i>Брайан Кэнтрилл</i>	
Глава 23. Распределенное программирование с MapReduce	416
<i>Джеффри Дин и Санджай Гхемават</i>	
Глава 24. Красота параллельной обработки	431
<i>Саймон Пейтон Джоунс</i>	
Глава 25. Синтаксическая абстракция: расширитель syntax-case	456
<i>P. Кент Дибиг</i>	
Глава 26. Архитектура, экономящая силы: объективно-ориентированная рабочая среда для программ с сетевой структурой	479
<i>Уильям Отте и Дуглас Шмидт</i>	
Глава 27. Объединение деловых партнеров с помощью RESTful	505
<i>Эндрю Патцер</i>	
Глава 28. Красивая отладка	518
<i>Андреас Целлер</i>	
Глава 29. Отношение к коду как к очерку	533
<i>Юкихиро Мацумото</i>	
Глава 30. Когда кнопка остается единственным предметом, связывающим вас с внешним миром	538
<i>Арун Мехта</i>	
Глава 31. Emacspeak: полноценно озвученный рабочий стол	560
<i>T. Раман</i>	
Глава 32. Код в развитии	586
<i>Лаура Уингерд и Кристофер Сейвалд</i>	
Глава 33. Создание программ для «Книги»	598
<i>Брайан Хэйес</i>	
Послесловие	612
О тех, кто работал над книгой	614

Содержание

Введение	16
Глава 1. Поиск соответствий с помощью регулярных выражений	17
<i>Брайен Керниган</i>	
Практика программирования	18
Глава 2. Дельта-редактор Subversion: Интерфейс и онтология	27
<i>Карл Фогель</i>	
Управление версиями и преобразование дерева	28
Выражение различий в деревьях каталогов	32
Интерфейс дельта-редактора	33
Глава 3. Самый красивый код, который я никогда не писал	46
<i>Джон Бентли</i>	
Мой самый красивый код	46
Усиление отдачи при сокращении размеров	48
Глава 4. Поиск вещей	60
<i>Тим Брэй</i>	
Фактор времени	60
Проблема: Данные веб-блога	60
Регулярные выражения	61
Подключение регулярных выражений к работе	62
Ассоциативное устройство хранения	66
А нужна ли оптимизация?	69
Проблема: Кто выбирал, что и когда?	70
Двоичный поиск	72
Сильные и слабые стороны двоичного поиска	74
Выход из цикла	75
Ищите в большом	76
Поиск с постингом	76
Ранжировка результатов	76
Поиск в Интернете	77
Вывод	78

Глава 5. Правильный, красивый, быстрый (именно в таком порядке): уроки разработки XML-верификаторов	79
<i>Элиот Расти Гарольд</i>	
Роль XML-проверки	79
Проблема	80
Версия 1: Простейшая реализация	82
Версия 2: Имитация BNF-нотации ценой $O(N)$ операций	83
Версия 3: Первая оптимизация $O(\log N)$	85
Версия 4: Вторая оптимизация: исключение двойной проверки	86
Версия 5: Третья оптимизация $O(1)$	88
Версия 6: Четвертая оптимизация: кэширование	93
Мораль всей этой истории	95
Глава 6. Платформа для проведения комплексного теста: красота, полученная за счет хрупкости конструкции	96
<i>Майкл Фезерс</i>	
Платформа для приемочных испытаний, выполненная в трех классах	97
Сложности конструкции платформы	99
Открытая платформа	101
Насколько просто может быть устроен HTML-парсер?	102
Вывод	105
Глава 7. Красивые тесты	107
<i>Альберто Савойя</i>	
Изрядно поднадоеvший двоичный поиск	108
Знакомство с JUnit	111
Подробный разбор двоичного поиска	113
Задымление разрешается (и приветствуется)	114
Проталкивание через границы	114
Элемент случайности в тестировании	118
Беспокойства о производительности	124
Вывод	126
Глава 8. Динамическая генерация кода для обработки изображений . .	128
<i>Чарльз Петцольд</i>	
Глава 9. Нисходящая иерархия операторов	152
<i>Дуглас Крокфорд</i>	
JavaScript	153
Таблица обозначений	154
Лексемы	155
Старшинство	156
Выражения	157
Инфиксные операторы	158

Префиксные операторы	160
Операторы присваивания	160
Константы	161
Область видимости	161
Операторы	163
Функции	166
Литералы массивов и объектов	168
Что нужно сделать и о чём подумать	169
Глава 10. Поиск методов ускоренного подсчета заполнения	170
<i>Генри Уоррен-мл.</i>	
Основные методы	171
«Разделяй и властвуй»	172
Другие методы	175
Сумма и разница подсчета заполнения двух слов	176
Сравнение подсчетов заполнений двух слов	177
Подсчет единичных битов в массиве	178
Применение	183
Глава 11. Безопасная связь: технология свободы	186
<i>Ашиш Гулхати</i>	
С чего все начиналось	187
Разбор проблем безопасного обмена сообщениями	189
Ключевая роль удобства и простоты использования	191
Основы	194
Конструктивные цели и решения	195
Конструкция основной системы	196
Блок тестирования	199
Функциональный прототип	200
Завершение, подключение, обкатка	201
Обновление хранилища электронной почты	202
Сохранность дешифрованной информации	204
Гималайский хакинг	205
Организация защиты кода	207
Ревизия Старт::GPG	208
Скрытые манипуляции	211
Скорость тоже имеет значение	213
Конфиденциальность связи для обеспечения прав человека	214
Хакинг цивилизации	215
Глава 12. Становление красивого кода в BioPerl	217
<i>Линкольн Стейн</i>	
BioPerl и модуль Bio::Graphics	218
Пример выходной информации Bio::Graphics	219
Требования, предъявляемые к Bio::Graphics	221
Процесс проектирования Bio::Graphics	223
Установка параметров	227
Выбор классов объектов	229
Обработка параметров	232

Пример кода	237
Динамические параметры	238
Расширение Bio::Graphics	242
Поддержка веб-разработчиков	243
Поддержка изображений типографского качества	244
Добавление новых глифов	245
Заключение и извлеченные уроки	247
Глава 13. Конструкция генного сортировщика	249
<i>Джим Кент</i>	
Пользовательский интерфейс программы Gene Sorter	250
Поддержание диалога с пользователем по Интернету	251
Небольшой полиморфизм может иметь большое значение	254
Фильтрация, оставляющая только значимые гены	257
Теория красоты кода в крупных формах	258
Вывод	262
Глава 14. Как первоклассный код развивается вместе с аппаратным обеспечением (на примере Гауссова исключения)	263
<i>Джек Донгарра и Петр Лушек</i>	
Влияние компьютерной архитектуры на матричные алгоритмы	264
Декомпозиционный подход	266
Простая версия	267
Подпрограмма DGEFA библиотеки LINPACK	269
LAPACK DGETRF	272
Рекурсивное использование LU	275
ScaLAPACK PDGETRF	278
Многопоточная обработка для многоядерных систем	283
Несколько слов об анализе ошибок и итоговом количестве операций	286
Дальнейшее направление исследований	287
Дополнительная литература	288
Глава 15. Долговременные выгоды от красивой конструкции	290
<i>Адам Колава</i>	
В чем, по-моему, заключается красота программного кода	290
Представление библиотеки лаборатории ЦЕРН	291
Внешняя красота	292
Внутренняя красота	298
Заключение	304
Глава 16. Модель драйверов ядра Linux: преимущества совместной работы	305
<i>Грег Кроа-Хартман</i>	
Скромное начало	306
Превращение в еще более мелкие части	311
Расширение масштаба до тысяч устройств	314
Свободно присоединяемые мелкие объекты	316

Глава 17. Иной уровень косвенного подхода	317
<i>Диомидис Спинеллис</i>	
От кода к указателям	317
От аргументов функций к аргументам указателей	320
От файловых систем к уровням файловой системы	324
От кода к предметно-ориентированному языку	326
Мультиплексирование и демультиплексирование	328
Уровни навсегда?	330
Глава 18. Реализация словарей Python: стремление	
быть всем во всем полезным	331
<i>Эндрю Кучлинг</i>	
Что делается внутри словаря	333
Специальные приспособления	335
Особая оптимизация для небольших хэшей	335
Когда специализация приводит к издержкам	335
Конфликтные ситуации	337
Изменение размера	338
Итерации и динамические изменения	339
Вывод	340
Благодарности	340
Глава 19. Многомерные итераторы в NumPy	341
<i>Трейвис Олифант</i>	
Основные сложности обработки N-мерных массивов	342
Модель памяти, занимаемой N-мерным массивом	343
Происхождение итератора NumPy	345
Конструкция итератора	346
Перемещение итератора	346
Завершение работы итератора	347
Настройка итератора	348
Определение показаний счетчика итератора	349
Структура итератора	350
Интерфейс итератора	352
Использование итератора	353
Итерация по всем, кроме одного измерения	354
Множественные итерации	355
Истории	357
Вывод	358
Глава 20. Высоконадежная корпоративная система, разработанная	
для миссии NASA Mars Rover	359
<i>Рональд Мак</i>	
Миссия и совместный информационный портал	360
Потребности миссии	361
Архитектура системы	363
Исследование конкретного примера: Служба информационных потоков	366

Глава 21. ERP5: Конструирование с целью достижения максимальной адаптивности	381
<i>Роджерио Эйтем де Карвальо и Рафаэль Моннерат</i>	
Общие цели ERP	382
ERP5	382
Базовая платформа Zope	384
Понятия, используемые в ERP5 Project	388
Программирование ERP5 Project	390
Вывод.	394
Глава 22. Ложка грязи.	396
<i>Брайан Кэнтрелл</i>	
Глава 23. Распределенное программирование с MapReduce	416
<i>Джеффри Дин и Санджай Гхемават</i>	
Пример мотивации	416
Модель программирования MapReduce	419
Распределенная реализация MapReduce	422
Расширения модели	426
Заключение	427
Дополнительная информация	428
Благодарности	428
Приложение: Решение задачи подсчета слов.	429
Глава 24. Красота параллельной обработки	431
<i>Саймон Пейтон Джоунс</i>	
Простой пример: банковские счета	432
Программная транзакционная память	435
Реализация транзакционной памяти	441
Задача Санта-Клауса	444
Основная программа	448
Размышления о языке Haskell	452
Вывод.	453
Благодарности	454
Глава 25. Синтаксическая абстракция: расширитель syntax-case	456
<i>P. Кент Дибиг</i>	
Краткое введение в syntax-case	460
Алгоритм расширения	463
Преобразование	474
Пример.	475
Вывод.	477
Глава 26. Архитектура, экономящая силы: объектно-ориентированная рабочая среда для программ с сетевой структурой	479
<i>Уильям Отте и Дуглас Шмидт</i>	

Типовое приложение: служба регистрации	481
Объектно-ориентированное проектирование рабочей среды	
сервера регистрации	484
Определение общих свойств	486
Увязка вариантов	487
Связывание воедино	489
Реализация последовательных серверов регистрации	492
Реактивный сервер регистрации	493
Реализация сервера регистрации с параллельной обработкой	497
Вывод	504
Глава 27. Объединение деловых партнеров с помощью RESTful	505
<i>Эндрю Патцер</i>	
Предыстория проекта	505
Предоставление служб внешним клиентам	506
Определение интерфейса службы	507
Маршрутизация службы с использованием шаблона Factory	510
Обмен данными с использованием протоколов электронного бизнеса	512
Вывод	517
Глава 28. Красивая отладка	518
<i>Андреас Целлер</i>	
Отладка отладчика	519
Системный подход к процессу отладки	521
Проблемы поиска	522
Автоматическое обнаружение причины отказа	523
Дельта-отладка	525
Минимизация входных данных	528
Проблема прототипа	531
Вывод	532
Благодарности	532
Дополнительная информация	532
Глава 29. Отношение к коду как к очерку	533
<i>Юкихиро Мацумото</i>	
Глава 30. Когда кнопка остается единственным предметом, связывающим вас с внешним миром	538
<i>Арун Мехта</i>	
Основная модель конструкции	539
Интерфейс ввода	543
Дерево	543
Длинный щелчок	544
Динамическое изменение заполнения дерева	548
Простой набор текста	549
Прогнозирование: завершение слова и следующее слово	550
Шаблоны и замена	551
Реализация кэш	552

Распространенные и избранные слова	553
Отслеживание путей	554
Буфер набора, редактирование и прокрутка	554
Буфер обмена	556
Поиск	557
Макрокоманды	557
Эффективность пользовательского интерфейса	558
Загрузка	558
Будущие направления	558

Глава 31. Emacspeak: полноценно озвученный рабочий стол 560*Т. Раман*

Создание речевого вывода	561
Говорящий Emacs	562
Генерация полноценного речевого вывода	564
Использование Aural CSS (ACSS) для стилевого оформления речевого вывода	569
Добавление звуковых обозначений	570
Беспроблемный доступ к интерактивной информации	574
Краткий отчет	581
Вывод	584
Благодарности	585

Глава 32. Код в развитии 586*Лаура Уингерд и Кристофер Сейвалд*

Применение «книгообразного» вида	587
Однаковое должно выглядеть однообразно	589
Опасность отступов	590
Перемещение по коду	591
Используемый нами инструментарий	592
Изменчивое прошлое DiffMerge	594
Вывод	596
Благодарности	597
Дополнительная информация	597

Глава 33. Создание программ для «Книги» 598*Брайан Хэйес*

Нелегкий путь	598
Предупреждение для тех, кто боится скобок	599
Три в ряд	600
Скользящий наклон	603
Неравенство в треугольнике	605
Блуждания по извилистостям	606
«Что вы говорите!» — то есть «Ara!»	608
Вывод	610
Дополнительная информация	610

Послесловие 612**О тех, кто работал над книгой** 614

Введение

Я начал работать программистом летом 1982 года. Через пару недель после этого один из системных администраторов дал мне почитать книги «*The Elements of Programming Style*», Brian W. Kernighan, P. J. Plauger, и «*Algorithms + Data Structures = Programs*», Niklaus Wirth.

Это стало для меня открытием. Я впервые узнал, что программы — нечто большее, чем простые инструкции для компьютера. Они могут быть элегантными, как ближайшее окружение президента, изящными, как подвесной мост, и красноречивыми, как эссе Джорджа Оруэлла.

С тех пор я неоднократно встречал людей, сокрушающихся о том, что никто не учит студентов видеть красоту в программировании. Архитекторы изучают постройки других архитекторов, композиторы — произведения, написанные их коллегами, но программисты обращаются к чужим программам с лишь одной целью — поиск и исправление ошибок. Мы непрерывно учим студентов давать переменным осмыслиенные имена, рассказываем об основных паттернах программирования, а потом удивляемся, почему их программы столь уродливы.

Эта книга — попытка исправить сложившуюся ситуацию. В мае 2006 года я попросил некоторых известных (и не очень) разработчиков программного обеспечения изучить и обсудить наиболее совершенные части программного кода, попадавшиеся им когда-либо на глаза. И, как показывает эта книга, они нашли красоту и совершенство во многих аспектах программирования. Для кого-то совершенство кроется в деталях искусно сработанного программного продукта; кому-то нравится «крупный план» — структура программы или приемы, использованные для воплощения задуманного.

Я благодарен всем, кто нашел время и принял участие в нашем проекте. Надеюсь, что вы получите от чтения этой книги такое же удовольствие, как Энди и я — от редактирования. Надеемся, что прочитанное вдохновит и вас на создание чего-нибудь совершенного.

Грег Уилсон

1 ПОИСК СООТВЕТСТВИЙ С ПОМОЩЬЮ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Брайен Керниган (*Brian Kernighan*)

Регулярные выражения — это система записей текстовых шаблонов, на основе которых фактически составлен специальный язык, предназначенный для поиска соответствий. Несмотря на существование несметного количества вариантов, все они следуют единому замыслу, при котором большинство символов шаблона соответствует своему собственному появлению, но некоторые *методы* имеют специальное назначение, к примеру, знак звездочки (*) указывает на некую повторяемость, или конструкция [...] означает любой (один) символ из набора, заключенного в квадратные скобки.

Практически большинство систем поиска в программах типа текстовых редакторов, предназначены для побуквенного поиска слов, то есть их регулярные выражения зачастую являются строкой литералов, наподобие `print`, которая соответствует любому из сочетаний: `rprintf`, или `sprintf`, или `printer paper`. В так называемых *символах шаблона*, используемых для обозначений имен файлов в Unix и Windows, знак звездочки (*) соответствует любому числу символов, поэтому шаблон `*.c` соответствует всем именам файлов, оканчивающихся на `.c`. Существует великое множество разнообразных регулярных выражений даже в тех областях, где обычно ожидается их однообразие. Джейфри Фридл (Jeffrey Friedl) в книге «Регулярные выражения» (Питер, 2003) представил подробное исследование на эту тему.

Стивен Клин (Stephen Kleene) изобрел регулярные выражения в середине 1950-х годов прошлого века как систему записей для вычислительных моделей конечных автоматов; по сути они и представляют собой эквивалент конечного автомата. Их первое появление в программных продуктах отмечено в середине 60-х годов, в разработанной Кеном Томпсоном (Ken Thompson) версии текстового редактора QED. В 1967 году Томпсон подал заявку на патент механизма быстрого поиска текстовых соответствий на основе применения регулярных выражений. В 1971 году этот патент был получен в числе самых первых патентов из категории программного обеспечения [U.S. Patent 3,568,156, Text Matching Algorithm, March 2, 1971].

Со временем регулярные выражения перекочевали из QED в текстовый редактор системы Unix под названием ed, а затем и в выдающийся Unix-инструмент grep, который Томпсон создал после радикального хирургического вмешательства в ed. Эти весьма популярные программы помогли познакомится с регулярными выражениями всему раннему Unix-сообществу.

Первоначальная версия созданного Томпсоном механизма поиска соответствий работала очень быстро, поскольку сочетала в себе две независимые идеи. Одна из них заключалась в создании машинных команд на лету, в процессе поиска, что позволяло системе работать не в режиме интерпретатора, а со скоростью самой машины. Вторая идея состояла в проведении поиска всех возможных соответствий на каждой стадии, чтобы не приходилось возвращаться и вести поиск других вариантов потенциальных соответствий. В следующих созданных Томпсоном текстовых редакторах, таких как ed, в коде поиска соответствий использовался простой алгоритм, предусматривающий по необходимости поиск с возвратом. Теоретически это замедляло процесс, но используемые на практике шаблоны редко требовали возврата, поэтому алгоритм и код ed и grep вполне подходили для решения большинства задач.

В последующие инструменты поиска соответствий, такие как egrep и fgrep, были добавлены более мощные классы регулярных выражений, а их механизмы были ориентированы на скоростную работу, не зависящую от характера используемых шаблонов. Некогда считавшиеся причудливыми регулярные выражения приобрели популярность и не только были включены в библиотеки, написанные на Си, но стали также частью синтаксиса таких языков сценариев, как Awk и Perl.

Практика программирования

В 1998 году мы с Робом Пайком (Rob Pike) работали над книгой по практическому программированию — «The Practice of Programming» (Addison-Wesley). В последней главе этой книги — «Notation» — приводилась подборка примеров того, как продуманная система записи кода ведет к созданию более совершенных программ и улучшает технологию программирования. В подборку было включено использование простого описания данных (например при использовании функции printf и при создании кода на основе таблиц).

С учетом наших давних связей с Unix, и почти 30-летнего опыта работы с инструментарием, основанным на записях регулярных выражений, нам захотелось включить в книгу обсуждение регулярных выражений, а также не обойти стороной и вопросы программной реализации их использования. В силу наших инструментальных предпочтений мы решили, что лучше сконцентрироваться на классе регулярных выражений, встречающемся в grep, а не на тех символах шаблона, которые используются, скажем, в системной оболочке, это к тому же дало бы возможность поговорить и о конструкции самого grep.

Проблема заключалась в том, что любой из существующих пакетов для работы с регулярными выражениями был для нас слишком велик. В частности,

в grep было свыше 500 строк (около 10 книжных страниц), имеющих разные, ненужные нам дополнения. Все пакеты регулярных выражений с открытым кодом страдали гигантоманией, занимая практически объем целой книги, и поскольку они разрабатывались с учетом универсальности, гибкости и быстродействия, ни один из них не подходил для учебных целей.

Я предложил Робу подыскать самый маленький пакет для работы с регулярными выражениями, на котором можно было бы проиллюстрировать основные идеи и преподать узнаваемые и полезные классы шаблонов. В идеале код должен был поместиться на одной странице.

Роб скрылся в своем офисе. Насколько я помню, ему понадобилась всего пара часов, чтобы впоследствии в главе 9 книги «The Practice of Programming» появились 30 строк кода, написанного на языке Си. В этом коде был реализован механизм поиска соответствий, обрабатывающий следующие конструкции.

Символ	Значение
C	Соответствует любой букве «C»
. (точка)	Соответствует любому одному символу
^	Соответствует началу входящей строки
\$	Соответствует концу входящей строки
*	Соответствует появлению предыдущего символа от нуля до нескольких раз

Это весьма полезный класс. Он запросто справляется с 95% моих повседневных потребностей в использовании регулярных выражений. Во многих ситуациях решение правильно сформулированной проблемы — довольно большой шаг навстречу созданию красивой программы. Роб заслуживает особой похвалы за то, что из имеющегося в его распоряжении весьма обширного набора средств он остановился на крайне небольшой, но все же важной, четко определенной и открытой для наращивания подборке функциональных возможностей.

Созданный Робом код сам по себе является превосходным примером красоты программирования: компактный, изящный, эффективный и полезный. Это один из лучших когда-либо встречавшихся мне примеров рекурсии, демонстрирующий всю мощь применения указателей в языке Си. Хотя в то время нас больше интересовало, как преподнести важность роли качественной записи кода, чтобы облегчить использование программы (а также, по возможности, упростить ее написание), код работы с регулярными выражениями помимо всего прочего стал превосходным примером создания алгоритмов, структуры данных, проведения тестирования, увеличения производительности и других важных моментов программирования.

Реализация

В книге «The Practice of Programming» код поиска соответствий регулярным выражениям является частью отдельной программы, имитирующей grep, но та часть кода, которая работает непосредственно с регулярными выражениями,

может быть полностью выделена из своего окружения. Основная программа особого интереса не представляет. Как и многие инструментальные средства Unix, она либо читает содержимое своего стандартного ввода, либо считывает последовательность файлов и выводит те строки, в которых встречаются соответствия регулярному выражению.

А код поиска соответствий выглядит следующим образом:

```
/* match: поиск соответствий регулярному выражению по всему тексту */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do { /* нужно просмотреть даже пустую строку */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: поиск соответствий регулярному выражению в начале текста */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: поиск регулярного выражения вида c* с начала текста */
int matchstar(int c, char *regexp, char *text)
{
    do { /* символ * соответствует нулю или большему количеству появлений */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

Обсуждение

Функция `match(regexp, text)` проверяет наличие в тексте фрагментов, соответствующих регулярному выражению; если соответствие найдено, она возвращает 1, а если нет — 0. Если соответствий более одного, она находит крайнее слева и самое короткое.

Основная операция поиска соответствий в функции `match` написана просто и понятно. Если в регулярном выражении первым идет символ ^ (обозначающий привязанное соответствие), любые возможные соответствия должны встречаться в начале строки. Таким образом, регулярному выражению ^xyz соответствует

только та последовательность `xuz`, которая встречается в самом начале строки, а не где-то в ее середине. Операция проверяет, соответствует ли остальной части регулярного выражения текст в начале строки, и нигде больше. Если символ привязки отсутствует, регулярному выражению может соответствовать фрагмент в любой части строки. Проверка ведется на соответствие шаблону в каждой последовательной символьной позиции текста. Если имеется несколько соответствий, будет определено только первое (самое левое) из них. Таким образом, регулярному выражению `xuz` будет соответствовать первое появление фрагмента `xuz`, независимо от того, где оно будет найдено.

Обратите внимание, что перемещение по входящей строке реализовано с помощью цикла `do-while`, сравнительно необычной конструкции для программ, написанных на Си. Появление `do-while` вместо `while` всегда будет вызывать вопрос: почему бы условия завершения цикла не проверить в его начале, вместо того чтобы проверять их в конце, когда уже что-то сделано? Но с проверкой здесь все в порядке: поскольку знак операции `*` допускает определение соответствия и при нулевой длине, сначала нужно проверить возможность данного нулевого соответствия.

Вся основная работа осуществляется в функции `matchhere`(`regexp, text`), проверяющей соответствие текста в текущей позиции регулярному выражению. Работа функции `matchhere` заключается в попытке найти соответствие первого символа текста первому символу регулярного выражения. Если попытка потерпит неудачу, значит, в данной текстовой позиции соответствие не найдено, и функция `matchhere` вернет 0. Если попытка будет удачной, значит, можно перемещаться к следующему символу регулярного выражения и следующему символу текста. Это делается с помощью рекурсивного вызова функции `matchhere`.

Ситуация усложняется некоторыми особыми случаями и, разумеется, необходимостью остановки рекурсии. Самый простой случай — окончание регулярного выражения (`regexp[0] == '\0'`), которое означает, что все предшествующие проверки прошли успешно и текст соответствует регулярному выражению.

Если регулярное выражение состоит из символа, сопровождаемого звездочкой (*), вызывается функция `matchstar`, проверяющая соответствие замкнутому выражению. Функция `matchstar`(`c, regexp, text`) пытается найти в тексте соответствие повторяющемуся символу `c`, начиная с нуля и т. д., до тех пор пока либо не будет найдено соответствие в остальной части текста, либо попытка окажется неудачной, что будет означать отсутствие соответствия. Этот алгоритм определяет «наименьшее соответствие», которое больше подходит для простых шаблонных соответствий, таких как в `grep`, когда все направлено на самый быстрый поиск. «Наибольшее соответствие» имеет более интуитивную природу и, скорее всего, больше подходит для текстовых редакторов, где соответствующий текст будет заменяться чем-нибудь другим. Самые современные библиотеки регулярных выражений предлагают оба альтернативных варианта, а в книге «The Practice of Programming» представлен простейший вариант `matchstar` для рассматриваемого далее случая.

Если регулярное выражение завершается символом \$, соответствие в тексте определяется лишь тогда, когда оно находится в самом его конце:

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

В противном случае, если мы не находимся в конце текстовой строки (то есть `*text != '\0'`) и если первый символ текстовой строки соответствует первому символу регулярного выражения, то это тоже неплохо; мы переходим к проверке соответствия следующего символа текста следующему символу регулярного выражения, осуществляя рекурсивный вызов функции `matchhere`. Эта рекурсия является сердцевиной алгоритма и основой компактности и ясности кода. Если все попытки поиска соответствий окажутся неудачными, значит, на данный момент между регулярным выражением и текстом соответствий нет, и функция `matchhere` возвращает 0.

В данном коде широко используются указатели языка Си. На каждом этапе рекурсии, если находится какое-нибудь соответствие, в следующем рекурсивном вызове используются арифметические операции над указателями (например `regexp+1` и `text+1`), чтобы в последующем функция вызывалась со следующим символом регулярного выражения и текста. Глубина рекурсии не превышает длины шаблона, который обычно достаточно невелик, чтобы не опасаться за выход за пределы пространства памяти.

Альтернативные варианты

Мы имеем дело с превосходным, написанным в отличном стиле фрагментом программного кода, но и он не может считаться абсолютным совершенством. Что в нем можно было бы изменить? Я бы перестроил функцию `matchhere`, чтобы сначала обрабатывался символ \$, а потом уже символ *. Хотя для данного случая это ничего не изменит, но будет выглядеть более естественно и соответствовать правилу, сначала обрабатывать более легкие варианты, а затем уже переходить к более сложным.

Вообще-то порядок проверки тоже имеет весьма важное значение. К примеру, в следующей проверке из функции `matchstar`:

```
} while (*text != '\0' && (*text++ == c || c == '.'));
```

мы вынуждены волей-неволей переместиться еще на один символ текстовой строки, поскольку инкрементное выражение `text++` должно быть обязательно выполнено.

Этот код обеспечивает условия завершения цикла. Как правило, успех соответствия определяется в том случае, если регулярное выражение заканчивается одновременно с текстом.

Если они заканчиваются вместе, это свидетельствует о соответствии; если одно из них заканчивается раньше другого, значит, соответствие отсутствует. Возможно, более очевидно это представлено в следующей строке:

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

но нестандартные условия завершения встречаются также и в других местах.

Версия функции `matchstar`, которая осуществляет поиск самого длинного соответствия с крайней левой позиции, начинается с определения максимальной последовательности появлений символа с во входящей строке. Затем, что-

бы попытаться продолжить поиск соответствий оставшейся части шаблона во всем остальном тексте, в ней используется вызов функции `matchhere`. Каждая неудача сокращает количество символов с на единицу и вызывает новую попытку, не исключая и случай нулевого появления этого символа:

```
/* matchstar: поиск самого длинного соответствия регулярному выражению c* с крайней левой позиции*/
int matchstar(int c, char *regexp, char *text)
{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do { /* * нуль и более соответствий */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

Рассмотрим регулярное выражение `(.*)`, которое соответствует любому тексту, взятому в скобки.

Применимельно к заданному тексту:

```
for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
```

самое длинное соответствие с самого начала будет определено как все выражение, взятое в скобки, а самое короткое соответствие остановится на первой же правой скобке. (Разумеется, самое длинное соответствие берет начало с другой, левой скобки и простирается до самого конца текста.)

Разработки на основе этого примера

Создавая книгу «The Practice of Programming», мы задавались целью научить хорошему стилю программирования. В те времена мы с Робом еще работали в компании *Bell Labs* и не имели своего собственного позитивного опыта использования материалов книги в учебном процессе. Поэтому нам было приятно узнать, что некоторые положения неплохо вписались в этот процесс. А с 2000 года я стал использовать данный программный код в качестве учебного пособия при обучении важным моментам программирования.

В первую очередь в этом примере показана польза применения рекурсий и их роль в создании качественного, нестандартного кода, который не является ни очередной версией *Quicksort* (или подпрограммы вычисления факториала), ни какой-либо разновидностью кода по перебору древовидной структуры.

К тому же этот пример может послужить неплохой основой для проведения экспериментов с производительностью. Его быстродействие не слишком отличается от скорости работы системной версии *grep*, что показывает на слишком большие затраты ресурсов при использовании рекурсивной технологии и что этот код не стоит улучшать.

С другой стороны, этот пример является неплохой иллюстрацией важности хорошо продуманного алгоритма. Если шаблон включает несколько последова-

тельностей `.*`, прямолинейная реализация потребует большого количества повторных проходов и в ряде случаев будет выполняться крайне медленно.

Стандартная Unix-утилита grep тоже содержит подобное свойство повторного прохода. К примеру, команда:

```
grep 'a.*a.*a.*a.a'
```

при обработке текстового файла объемом в 4 Мб на обычной машине выполняется около 20 секунд.

Реализация, основанная на преобразовании недетерминированного конечно-го автомата к детерминированному, как в egrep, будет обладать намного лучшей производительностью в наиболее сложных случаях; она может провести обработку при таком же шаблоне и том же объеме текста менее чем за одну десятую секунды, и вообще, время ее работы от шаблона не зависит.

Расширения класса регулярных выражений могут стать основой для разнообразных вариантов их применения.

Например:

1. Добавление других метасимволов, таких как `+` для одного или нескольких появлений предыдущего символа или `?` для нулевого или одинарного соответствия. Добавление какого-нибудь способа ссылок на метасимволы, например, `\$` для случаев лiteralного появления символа `$`.
2. Разделение обработки регулярных выражений на фазу *компиляции* и фазу *выполнения*. Компиляция преобразует регулярное выражение во внутреннюю форму, которая упрощает код поиска соответствий или способствует более быстрому поиску последовательности соответствий. Для простейшего класса оригинально заданных регулярных выражений такое разделение не потребуется, но оно имеет смысл для приложений типа grep, где используется класс побогаче и одни и те же регулярные выражения применяются для обработки большого количества входящих строк.
3. Добавление символьных классов, таких как `[abc]` и `[0-9]`, которые в системе записей, принятых в grep, соответствуют `a`, или `b`, или `c` и цифровым символам соответственно. Это можно сделать несколькими способами, наиболее естественный из которых представляется в виде замены в исходном коде переменных `char*` структурами:

```
typedef struct RE {
    int type; /* CHAR, STAR, и т. д. */
    int ch; /* символ как таковой */
    char *ccl; /* для замены [...]*/
    int ncc1; /* для инвертированного класса [^...] */
} RE;
```

и модификации базового кода для обработки этих массивов взамен обработки символьных массивов. В данном случае отделять компиляцию от выполнения нет особой необходимости, но оказывается, это во многом все упрощает. Студенты, последовавшие совету по проведению предварительной компиляции подобной структуры, добиваются неизменно больших успехов, чем те, которые пытаются сходу интерпретировать некоторые сложные шаблонные структуры данных.

Создание понятных и однозначных спецификаций для символьных классов занятие непростое, а их достойное осуществление — и того сложнее и требует утомительного программирования. Со временем я упростил эту задачу, и теперь чаще всего прошу использовать сокращения, подобные тем, что используются в Perl, например \d для цифровых символов и \D для нецифровых, взамен использовавшихся ранее диапазонов символов, заключенных в скобки.

4. Использование абстрактного типа (opaque type) для скрытия RE-структуры и всех деталей ее реализации. Это довольно неплохой способ демонстрации всей сущности объектно-ориентированного программирования на Си, которое не поддерживает многое из того, что лежит за пределами этой технологии. В результате будет создан класс регулярных выражений, использующий для методов такие имена функций, как RE_new() и RE_match(), взамен синтаксическим украшениям объектно-ориентированного языка.
5. Изменение класса регулярных выражений путем уподобления его групповым символам различных системных оболочек со следующими признаками: соответствие имеют неявную привязку к обоим концам строки, символ * соответствует любому количеству символов, а символ ? соответствует любому отдельному символу. Можно изменить алгоритм или приспособить входные данные под существующий алгоритм.
6. Перенос кода на язык Java. В исходном коде довольно неплохо используются указатели Си, но поиск альтернатив в различных языках станет весьма неплохой практикой программирования. В версиях Java используется либо метод String.charAt (индексирование вместо указателей), либо метод String.substring (более похожий на версию с использованием указателей). Ни один из них не кажется столь же понятным, как Си-код, и ни один не обладает такой же компактностью. Поскольку производительность не входит в планы этого упражнения, интересно будет посмотреть, как код, реализованный на языке Java, будет работать раз в шесть-семь медленнее по сравнению с версиями, реализованными на языке Си.
7. Создание интерфейсного класса (wrapper class), который превращает регулярные выражения этого класса в имеющиеся в Java классы Pattern и Matcher, которые разделяют компиляцию и поиск соответствий совершенно различными способами. Это хороший пример использования шаблонов Adapter или Facade, которые придают различный облик существующим классам или наборам функций.

Я также широко использовал этот код для исследования технологий тестирования. Регулярные выражения обладают достаточно широкими возможностями, чтобы превратить тестирование в довольно необычный процесс, но их возможностей не хватает для того, чтобы каждый смог быстро написать солидную подборку автоматически исполняемых тестов. Для расширений, похожих на только что перечисленные, я просил студентов написать большое количество тестов компактным языком (еще один пример продуманной системы записи) и испробовать эти тесты на своем собственном коде; естественно, я испытывал их тесты на коде, созданном другими студентами.

Вывод

Когда Роб Пайк создал этот код, я был поражен его компактностью и великолепием – он был намного короче и мощнее, чем я вообще мог себе представить. Теперь, по прошествии времени, любой может указать на ряд причин, по которым этот код получился столь компактным.

Во-первых, благодаря хорошей подборке возможностей, направленных на извлечение максимальной пользы и на то, чтобы реализация была максимально понятна и не содержала в себе каких-либо изощрений. К примеру, для реализации привязанных шаблонов `^` и `$` потребовалось всего три или четыре кодовые строчки, но в них ясно показано, как справиться с особыми случаями, перед тем как проводить однотипную обработку общих случаев. Нужно было представить замкнутую операцию `*`, поскольку она входит в фундаментальные понятия регулярных выражений и обеспечивает единственный способ обработки шаблонов неопределенной длины. А вот обработка символов `+` и `?` ничего бы не добавила к пониманию сути вопроса, поэтому она была оставлена для использования в качестве упражнения.

Во-вторых, выигрыш был за счет применения рекурсии. Эта основная технология программирования почти всегда приводит к более компактному, понятному и изящному коду, чем его эквиваленты, использующие циклы в явном виде; именно она и была применена в данном случае. Идея брать по одному символу для проверки соответствия из начала регулярного выражения и из текста, а затем проводить рекурсивный вызов функции для всего остального является отголоском рекурсивной структуры примеров традиционного вычисления факториала или длины строки, но в более интересном и полезном воплощении.

В-третьих, в этом коде для достижения максимального эффекта используются все преимущества исходного языка. Разумеется, указатели могут применяться и не по назначению, но здесь они использованы для создания компактных выражений, которые естественным образом изображают извлечение отдельных символов и перемещение к следующему символу. Такого же эффекта можно достичь с помощью индексов массива или извлечения подстрок, но в данном коде указатели лучше справляются с работой, особенно в сочетании с идиомами Си, используемыми для автоприращения и неявного преобразования истинных значений.

Мне незнаком какой-либо другой пример программного кода, который проделывает столь же существенную работу при таком же скромном количестве строк, наряду с предоставлением столь богатого источника понятий и передовых идей.

2 Дельта-редактор Subversion: Интерфейс и онтология

Карл Фогель (Karl Fogel)

Почему-то в качестве примеров идеального кода чаще всего приводятся решения каких-то частных, четко очерченных и легко формулируемых проблем, таких как конструкция Даффа – Duff's Device ([http://en.wikipedia.org/wiki/Duff's_device](http://en.wikipedia.org/wiki/Duff%27s_device)) или алгоритм вычисления контрольной суммы – rsync rolling checksum algorithm (<http://en.wikipedia.org/wiki/Rsync#Algorithm>). Причина кроется не в том, что к идеальным можно отнести лишь небольшие и несложные решения, просто для восприятия сложного кода тот объем контекста, который способен поместиться на обратной стороне салфетки, явно недостаточен.

Получив в свое распоряжение несколько страниц, я хочу рассказать о другой, более объемной разновидности идеального кода. Это та категория решений, которые не сразу бросаются в глаза случайному читателю, но могут быть по достоинству оценены профессиональными программистами, поскольку в них аккумулируется опыт решения проблем в какой-то определенной области. Приводимый мною пример относится не к алгоритму, а к интерфейсу: программному интерфейсу, использующемуся для Subversion (<http://subversion.tigris.org>) – системы управления версиями с открытым кодом, для выражения различий между двумя деревьями каталога, который к тому же является и интерфейсом, использующимся для преобразования одного дерева в другое. В Subversion этот интерфейс носит формальное название Си-типа `svn_delta_editor_t`, но в просторечии он известен как *дельта-редактор* (delta editor).

В дельта-редакторе Subversion проявляются именно те свойства, которые ожидаются программистами от хороших конструкций. В нем настолько естественно проведено разграничение проблем, что любой разработчик новой функции Subversion легко может определить, когда и зачем нужно вызывать каждую функцию. Он дарит программисту возможность не ломая голову достичь максимальной эффективности (например за счет снижения ненужной передачи данных по сети) и предусматривает простую интеграцию вспомогательных задач (таких как сообщение о состоянии работ). Но, наверное, самое главное,

что конструкция доказала свою жизнестойкость в процессе расширений и обновлений.

И как будто в подтверждение интуитивных предположений о том, откуда берутся хорошие конструкции, дельта-редактор был создан всего лишь одним человеком в течение нескольких часов (хотя надо отметить, что этот человек хорошо знал суть проблемы и основы ее программной реализации).

Чтобы понять, в чем, собственно, заключается красота дельта-редактора, нужно приступить к изучению решаемых им проблем.

Управление версиями и преобразование дерева

На ранней стадии существования проекта Subversion команда поняла, что имеет дело с общей задачей, которая будет многократно выполняться: в чем выражаются минимальные различия между двумя похожими (обычно взаимосвязанными) деревьями каталога. Поскольку Subversion является системой управления версиями, одна из ее задач состоит в отслеживании изменений, вносимых в структуру каталогов, а также в содержимое отдельных файлов. Фактически в основу конструкции хранилища Subversion, находящего на сервере, заложено управление версиями каталогов. Хранилище представляет собой простую серию зафиксированных состояний дерева каталогов в процессе преобразования этого дерева с течением времени. Для каждого изменения, переданного в хранилище, создается новое дерево, которое отличается от предыдущего дерева лишь там, где произошли изменения, и нигде больше. Неизменившиеся фрагменты нового дерева совместно используют хранилище с прежним деревом, и то же самое происходило и в прошлом. Каждая успешная версия дерева маркирована постоянно возрастающим целым числом – уникальным идентификатором, который называется *номером версии*.

Хранилище можно представить массивом номеров версий, простирающихся в бесконечность. По соглашению нулевая версия всегда представляет собой пустой каталог. На рис. 2.1 версия 1 имеет дерево, свисающее из этого пустого каталога (обычно она представляет собой начальный импорт содержимого в хранилище), и на этот момент времени никакие другие версии еще не были переданы. В этой виртуальной файловой системе узлы отображены прямыми угольниками: каждый узел это либо каталог (помеченный в левом верхнем углу как DIR), либо файл (помеченный как FILE).

Что же происходит, когда мы модифицируем каталог `tuna`? Сначала мы создаем новый файловый узел, в котором содержится самый последний текст. И пока этот новый узел еще ни с чем не связан. На рис. 2.2 показано, что он зависит где-то в пространстве без собственного имени.

Затем мы создаем новую версию его родительского каталога. На рис. 2.3 показано, что подграф все еще не связан с массивом версий.

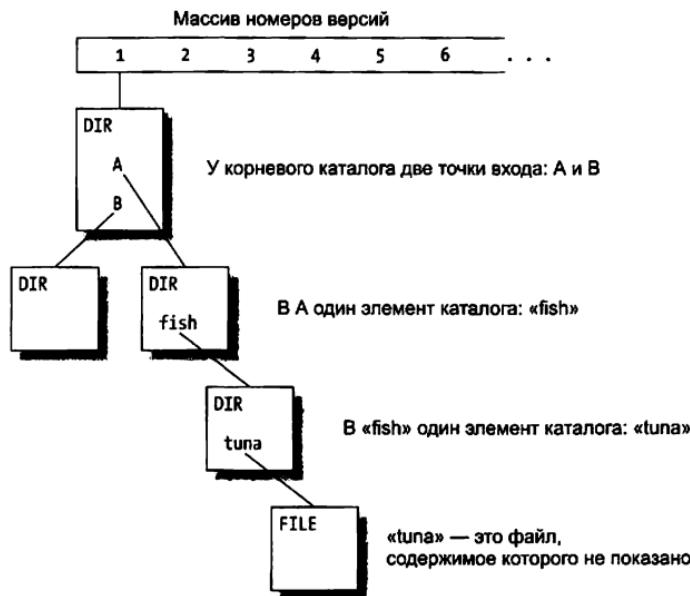


Рис. 2.1. Концептуальное представление номеров версий



Рис. 2.2. Новый, только что созданный узел

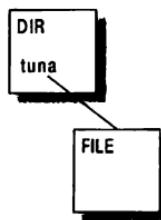


Рис. 2.3. Создание нового родительского каталога

Мы продолжаем линию, создавая новую версию следующего родительского каталога (рис. 2.4).

На рис. 2.5 показано, как на самом верху мы создаем новую версию корневого каталога. Этому новому каталогу нужна точка входа в «новый» каталог А, но поскольку каталог В вообще не претерпел никаких изменений, новый корневой каталог также имеет вход, указывающий на *старый* каталог узла В.

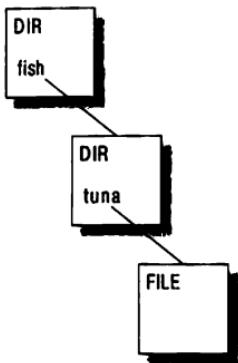


Рис. 2.4. Продолжение движения вверх, создание родительских каталогов

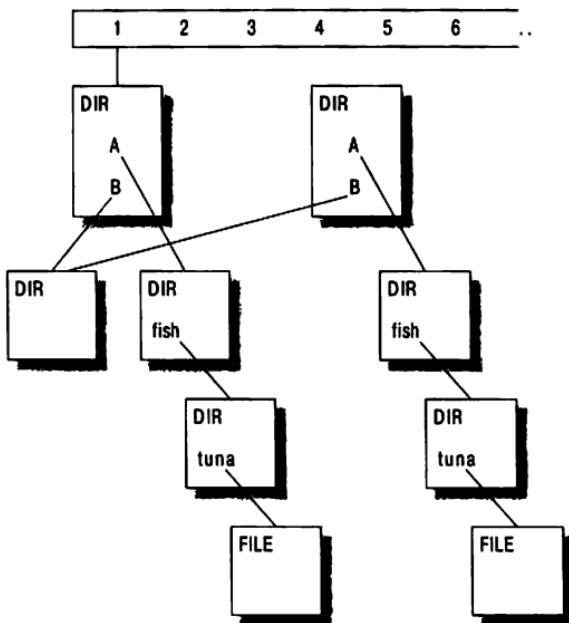


Рис. 2.5. Новое дерево каталогов в завешенном виде

Теперь, когда записаны все новые узлы, мы завершаем процесс «всплытия» наверх, привязывая новое дерево к следующей доступной версии в историческом массиве, делая его видимым для пользователей хранилища (рис. 2.6). В данном случае новое дерево становится версией 2.

Таким образом, каждая версия в хранилище указывает на корневой узел уникального дерева, и различия между этим деревом и предыдущим заклю-

чаются в изменениях, переданных в новой версии. Для отслеживания изменений программа одновременно спускается по двум деревьям, отмечая, где входы указывают на разные места. (Для краткости я опустил некоторые детали, например экономию пространства хранилища за счет сжатия старых узлов, что отличает их от тех же новых версий.)

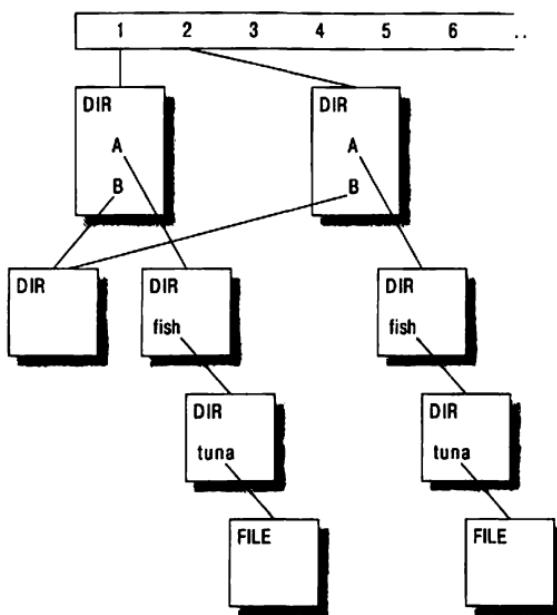


Рис. 2.6. Законченная версия: связь с новым деревом установлена

Хотя эта модель версий, представленных в древовидной структуре, служит лишь фоном для основного вопроса данной главы (дельта-редактора, к рассмотрению которого мы скоро приступим), у нее есть ряд замечательных свойств, которые я посчитал бы вполне достойными рассмотрения в отдельной главе, в качестве примера идеального кода. Вот некоторые из этих замечательных свойств.

Содержимое легко читается

Чтобы найти версию n файла `/path/to/foo.txt`, нужно перейти к версии n и спуститься вниз по дереву к файлу `/path/to/foo.txt`.

Записывающие и читающие не мешают друг другу

В то время, как записывающие создают новый узел, постепенно прокладывая путь наверх, для читающих сам процесс работы не виден. Новое дерево становится видимым читателям лишь после того, как записывающие прокладывают его последнюю связь к номеру версии в хранилище.

Создаются новые версии древовидной структуры

Структура каждого дерева сама по себе сохраняется от версии к версии. Переименование, добавление и удаление файла и каталога становится внутренней частью истории хранилища.

Если бы Subversion представляла собой одно лишь хранилище, то на этом история и закончилась бы. Но у этой системы есть еще и клиентская сторона: *рабочая копия*, которая представляет собой выбранную пользователем копию дерева какой-нибудь версии, дополненную всеми локальными правками, внесеными, но еще не переданными пользователем. (На самом деле рабочие копии не всегда являются отражением единственного дерева версии; зачастую они содержат некое смешение узлов, взятых из различных версий. Но поскольку мы рассматриваем вопрос преобразования дерева, то для нас это несущественно. Поэтому, сообразуясь с целями, поставленными в данной главе, будем считать, что рабочая копия представляет собой некое дерево, которое необязательно должно относиться именно к последней версии.)

Выражение различий в деревьях каталогов

Наиболее частым действием в Subversion является передача изменений, существующих между двумя сторонами: из хранилища в рабочую копию при обновлениях, для получения изменений, внесенных другими людьми, и из рабочей копии в хранилище, при передаче персонально внесенных изменений. Выражение различий между двумя деревьями является также ключом ко многим другим операциям общего порядка — например к анализу изменений, переходам по ветвлениям, объединением изменений из одной ветви в другую и т. д.

Очевидно, что иметь два разных интерфейса, один для направления сервер \rightarrow клиент, а другой для направления клиент \rightarrow сервер — было бы глупо. В обоих случаях основная задача одна и та же. Различия в деревьях таковыми и остаются, независимо от того направления, по которому они путешествуют в сети, или от того, что их потребитель намеревается с ними сделать. Но поиски естественного способа для выражения различий в деревьях неожиданно оказались совершенно не простой задачей. К тому же усложняло задачу и то обстоятельство, что Subversion поддерживает несколько сетевых протоколов и много-машинные механизмы хранения данных; то есть мы нуждались в интерфейсе, который бы выглядел одинаково, невзирая на все эти особенности.

Наши первые попытки придумать интерфейс колебались от неудовлетворительных до явно неуклюжих. Я не стану здесь рассказывать о всех этих попытках, но у них была одна общая тенденция — оставлять нерешенными вопросы, на которые не было никаких убедительных ответов. Например, многие решения предусматривали передачу изменившихся структурных путей в виде строк, а не в виде полных путей или их компонентов. Тогда в какой последовательности нужно передавать эти пути? Что передавать сначала — глубину или ширину? Может быть, передавать все в случайном порядке или в алфавитном? Должны ли отличаться команды для каталогов от команд для файлов? Самый важный вопрос заключался в том, каким образом каждая отдельная команда,

выражающая различия, будет знать, что она являлась частью более масштабной операции, группирующей все изменения в единый набор? В Subversion понятие завершенной операции с деревом каталогов предполагает, что он станет полностью виден со стороны пользователя и, если программный интерфейс по своим свойствам не отвечает этому понятию, то компенсация его недостатков требует создания уймы ненадежных программных заплаток.

В расстроенных чувствах я отправился с другим разработчиком, Беном Коллинз-Сассманом (Ben Collins-Sussman), из Чикаго на юг, в город Блюмингтон, штат Индиана, за советом к Джиму Блэнди (Jim Blandy), который, во-первых, был изобретателем модели хранилища Subversion, и во-вторых, имел, если можно так выразиться, стойкие взгляды на его конструкцию. Джим молча выслушал наш рассказ о различных направлениях, которые мы исследовали для передачи различий в деревьях каталогов, и по ходу нашего повествования становился все более мрачным. Когда мы достигли конца нашего списка, он посидел еще минутку, а затем вежливо попросил нас на некоторое время удалиться, чтобы не мешать ему думать. Я обул кроссовки и отправился на пробежку; Бен остался в другой комнате, почитать книгу или заняться чем-нибудь еще. Так, для общего развития.

После того как я вернулся с пробежки и принял душ, мы с Беном вернулись в комнату Джима, и он показал нам то, что придумал. По сути это была система Subversion в ее сегодняшнем представлении; с годами она подверглась различным изменениям, но фундаментальная структура осталась неизменной.

Интерфейс дельта-редактора

Все, что представлено далее, является слегка сокращенной версией интерфейса дельта-редактора. Я опустил те его части, которые занимаются копированием и переименованием, имеют отношение к свойствам Subversion (имеются в виду те из них, которые создают метаданные версий и не представляют ценности для рассмотрения), а также те части, которые занимаются некоторой другой «бухгалтерией» Subversion. Но вы всегда можете посмотреть на последнюю версию дельта-редактора, посетив веб-сайт <http://svn.collab.net/repos/svn/trunk/subversion/include/>

`svn_delta.h`

В данной главе рассматривается версия r21731 – http://svn.collab.net/viewvc/svn/trunk/subversion/include/svn_delta.h?revision=21731.

Чтобы понять работу интерфейса даже при допущенных сокращениях, нужно знать жargon, используемый в Subversion.

`pools (пулы)`

Аргументы `pool` – это пулы памяти, то есть распределенные буферы, позволяющие одновременно находиться в свободном состоянии большому количеству объектов.

`svn_error_t`

Возвращение ошибки типа `svn_error_t` означает, что функция возвращает указатель на объект ошибки Subversion; при успешном вызове возвращается нулевой указатель (`null`).

text delta (текущие различия)

Понятие *text delta* означает различия между двумя разными версиями файла; к *text delta* можно относиться как к тому исправлению, которое следует внести в одну версию файла, чтобы получить его другую версию. В Subversion в качестве «текстового» (*text*) файла рассматриваются двоичные данные, независимо от того что файл сам по себе может быть обычным текстом, аудиоданными, изображением или чем-нибудь еще. Текущие изменения (*text delta*) выражаются как потоки окон фиксированного размера. В каждом окне содержится порция отличающихся двоичных данных. При этом самый большой используемый участок памяти пропорционален размеру одного окна, а не всему размеру исправления (которые, если говорить о файле изображения, могут иметь довольно большой размер).

window handler (обработчик окна)

Это функциональный прототип, позволяющий применить одно окно текстовых изменений к целевому файлу.

baton (эстафетная палочка, маркер)

Это структура данных типа *void **, которая предоставляет контекст функции обратного вызова. В других API она иногда называется *void *ctx*, *void *userdata* или *void *closure*. В Subversion ее называют эстафетной палочкой — «*baton*», поскольку она служит маркером, который многократно передается по кругу, как палочка в эстафете.

Интерфейс начинается с введения, чтобы направить мышление читателя кода в правильное русло. Этот текст почти не изменился с тех пор, как Джим Блэнди написал его в августе 2000 года, а это означает, что основная концепция выдержала испытание временем:

```
/** Перемещение изменений дерева.
 *
 * В Subversion мы имеем дела с различными создателями и потребителями
 * изменений дерева.
 *
 * При обработке команды `commit':
 * - клиент проверяет свою рабочую копию данных и создает дельту дерева.
 * описывающую передаваемые изменения.
 * - сетевая библиотека клиента потребляет эти изменения (дельту) и посыпает
 * их по сети в виде эквивалентных серий сетевых запросов.
 * - сервер принимает эти запросы и создает дельту дерева ---
 * надеемся, эквивалентную той, что ранее была создана клиентом.
 * - модуль сервера Subversion забирает эту дельту и осуществляет
 * соответствующую транзакцию в адрес файловой системы
 *
 * При обработке команды `update' процесс происходит в обратном порядке:
 * - модуль сервера Subversion обращается к файловой системе и создает дельту
 * дерева, описывающую изменения, которые необходимо внести, чтобы обновить
 * клиентскую рабочую копию.
 * - сервер забирает эту дельту и производит сборку ответа, представляющего
 * соответствующие изменения.
 * - сетевая библиотека клиента принимает этот ответ и создает дельту дерева.
 * надеемся, эквивалентную той, что ранее была создана сервером Subversion.
```

* - библиотека рабочей копии забирает эту дельту и вносит соответствующие
* изменения в рабочую копию.
*
* Проще всего было бы представить дельту дерева, используя очевидную
* структуру данных. Чтобы провести обновление, сервер должен создать дельта-
* структуру, а библиотека рабочей копии должна применить эту структуру к
* рабочей копии; на сетевом уровне вся работа должна сводиться к простому
* переносу структуры по сети в неповрежденном виде.
*
* Но ожидается, что эти дельты временами будут слишком большими, чтобы
* поместиться в буфер обмена обычной рабочей станции. Например, при
* извлечении исходного дерева объемом 200Mb все оно представляется в виде
* одной дельты дерева. Поэтому важно обеспечить обработку таких дельт,
* которые слишком велики, чтобы целиком поместиться в буфер обмена.
*
* Поэтому вместо полного представления дельты дерева мы определяем
* для потребителя стандартный способ обработки каждой порции дельты дерева
* по мере ее создания производителем. Структура `svn_delta_editor_t`
* представляет собой набор функций обратного вызова, определяемых
* потребителем дельты и вызываемых ее производителем. В каждом вызове
* обратной функции описывается порция дельты --- изменения содержимого
* файла, частных переименований и т. д.
*/

Затем следует длинный комментарий, относящийся к официальной документации, за которым следует сам интерфейс, представляющий собой таблицу обратных вызовов с частично предопределенным порядком:

```
/** Структура, заполненная функциями обратного вызова, вызываемыми  
* источником дельты по ее готовности.  
*  
* Использование функций  
* —————  
*  
* Описание порядка использования функций для выражения дельты дерева.  
*  
* Дельта-потребитель приводит в исполнение описываемые в этой структуре  
* функции обратного вызова, а дельта-производитель их вызывает. Поэтому  
* вызывающая сторона (производитель) проталкивает данные дельты дерева  
* вызывающему (потребителю).  
*  
* В начале их прохождения потребитель предоставляет общий для всего  
* редактирования дельты маркер edit_baton.  
*  
* Затем, если в выражении изменений есть какая-нибудь дельта дерева,  
* производитель должен передать edit_baton функции open_root, чтобы  
* получить маркер (baton), представляющий корневой каталог  
* редактируемого дерева.  
*  
* Назначение большинства функций обратного вызова не вызывает сомнений:  
*  
* delete_entry – удаление элемента  
* add_file – добавление файла  
* add_directory – добавление каталога  
* open_file – открытие файла  
* open_directory – открытие каталога  
*/
```

- * Каждой из функций передается маркер, указывающий на тот каталог.
- * в котором произошли изменения, и аргумент `path`, дающий путь к
- * файлу, содержимому подкаталога или каталога, подлежащему изменениям
- * (относительно редактируемого корневого каталога). Редакторы обычно будут
- * требовать объединения этого относительного пути с каким-то базовым
- * путем, сохраненным в маркере редактирования (например URL, местоположения
- * в файловой системе операционной системы).
- *
- * Поскольку для каждого вызова требуется маркер родительского каталога,
- * включая и вызовы функций `add_directory` и `open_directory`, откуда все-таки
- * берется исходный маркер каталога, позволяющий запустить весь процесс?
- * Функция `open_root` возвращает маркер самого верхнего изменяемого каталога.
- * В общем, производитель изменений, прежде чем что-то сделать, должен вызвать
- * редакторскую функцию `open_root`.
- *
- * Наряду с тем что `open_root` предоставляет маркер корневого каталога
- * подвергаемого изменениям дерева, функции обратного вызова `add_directory`
- * и `open_directory` предоставляют маркеры других каталогов. Подобно
- * вышеупомянутым функциям обратного вызова, они используют маркер
- * родительского каталога, `parent_baton`, и относительный путь, а затем
- * возвращают новый, дочерний маркер для создаваемого или изменяемого
- * каталога – `child_baton`. Затем производитель может использовать этот маркер
- * `child_baton` для производства последующих изменений в этом подкаталоге.
- *
- * Таким образом, если уже существуют подкаталоги с именами 'foo' и
- * 'foo/bar', то производитель может создать новый файл под названием
- * 'foo/bar/baz.c', вызвав следующие функции:
- *
- * - `open_root()` – получение корневого маркера каталога верхнего уровня
- *
- * - `open_directory(root, "foo")` – получение маркера f для 'foo'
- *
- * - `open_directory(f, "foo/bar")` – получение маркера b для 'foo/bar'
- *
- * - `add_file(b, "foo/bar/baz.c")`
- *
- * Когда производитель завершает внесение изменений в каталог, он должен
- * вызвать функцию `close_directory`. Это позволяет потребителю совершать любые
- * необходимые зачистки и освободить хранилище маркеров.
- *
- * При вызове функций `add_file` и `open_file`, каждая из них возвращает
- * маркер для создаваемого или изменяемого файла. Этот маркер затем может
- * быть передан функции `apply_textdelta` для внесения изменений в содержимое
- * файла.
- * Когда производитель завершает внесение изменений в файл, он должен вызвать
- * функцию `close_file`, чтобы позволить потребителю произвести зачистку
- * и освободить маркер.
- *
- * Порядок вызова функций
-
- *
- * На порядок, в котором производитель может использовать маркер.
- * накладывается пять ограничений.
- *
- * 1. Производитель может вызывать функции `open_directory`, `add_directory`,
- * `open_file`, `add_file` для любого данного элемента каталога не более одного

```
* раза. Функция delete_entry может быть вызвана для любого данного элемента
* каталога не более одного раза, и позже за этим вызовом может следовать для
* этого же элемента каталога вызов функции add_directory или add_file.
* Функция delete_entry не может быть вызвана для любого элемента каталога
* после того, как для этого же элемента были вызваны функции
* open_directory, add_directory, open_file или add_file.
*
* 2. Производитель не может закрыть маркер каталога до тех пор, пока не
* закрыл все маркеры для всех его подкаталогов.
*
* 3. Когда производитель вызывает функцию open_directory или функцию
* add_directory, для них должны быть определены самые последние открытые
* маркеры, относящиеся к текущему открытому каталогу. Иными словами,
* производитель не может иметь одновременно два открытых маркера для
* разветвляющихся родственных элементов.
*
* 4. Когда производитель вызывает функцию open_file или функцию add_file, то
* перед появлением других вызовов файла или каталога она должна последовать
* любым внесенным в файл изменениям (используя функцию apply_textdelta).
* а затем вызвать функцию close_file.
*
* 5. Когда производитель вызывает функцию apply_textdelta, то перед
* осуществлением любых других вызовов svn_delta_editor_t, она должна
* произвести вызовы всех обработчиков окон (включая в конечном итоге
* и NULL-окно).
*
* Итак, производителю нужно использовать маркер каталогов и файлов в режиме
* единственного прохода дерева в глубину.
*
* Использование пулов
* -----
*
* Многие функции редактора вызываются по несколько раз в той
* последовательности, которая определяется "драйвером" редактора. Этот
* драйвер отвечает за создание пула, используемого для каждого повторного
* вызова функции редактора, и за очистку этого пула между вызовами. Драйвер
* передает соответствующий пул при каждом вызове функции.
*
* На основании требования о вызове функций редактора в режиме "только
* в глубину" драйвер обычно проводит вложение пулов в соответствии со
* структурой. Однако это всего лишь мера предосторожности, обеспечивающая
* безусловную очистку пулов, связанных с самыми глубокими записями, вместе
* с очисткой записей верхнего уровня. В интерфейсе не предполагается и не
* требуется какой-нибудь особенной организации пулов, передаваемых этими
* функциям.
*/
typedef struct svn_delta_editor_t
{
    /** Установка *root_baton в качестве маркера для верхнего изменяемого
     * каталога.
     * (Это вершина изменяемой ветви, необязательно корневой каталог
     * файловой системы.) В отношении root_baton производитель при завершении
     * изменений должен вызвать функцию close_directory, как и в случае
     * с любым другим маркером каталога.
     */
    svn_error_t *(*open_root)(void *edit_baton,
```

```
apr_pool_t *dir_pool,
void **root_baton);
/** При удалении элемента каталога, названного path, дочерний элемент
 * каталога представляется маркером parent_baton.
 */
svn_error_t *(*delete_entry)(const char *path,
                             void *parent_baton,
                             apr_pool_t *pool);
/** Мы собираемся добавить новый подкаталог, названный path. Мы будем
 * использовать значение, сохраняющееся функцией обратного вызова в
 * child_baton, в качестве parent_baton для дальнейших изменений в новом
 * подкаталоге.
 */
svn_error_t *(*add_directory)(const char *path,
                             void *parent_baton,
                             apr_pool_t *dir_pool,
                             void **child_baton);
/** Мы собираемся внести изменения в подкаталог (каталога,
 * идентифицированного маркером parent_baton). Подкаталог указан в
 * path. Обратный вызов функции должен привести к сохранению значения
 * в маркере *child_baton, который должен использоваться в качестве
 * parent_baton для последующих изменений в этом подкаталоге.
 */
svn_error_t *(*open_directory)(const char *path,
                             void *parent_baton,
                             apr_pool_t *dir_pool,
                             void **child_baton);
/** Мы завершили обработку подкаталога, имеющего маркер dir_baton
 * (установленный add_directory или open_directory). Мы больше не
 * собираемся использовать этот маркер, поэтому ресурсы, на которые он
 * ссылается, теперь могут быть освобождены.
 */
svn_error_t *(*close_directory)(void *dir_baton,
                               apr_pool_t *pool);
/** Мы собираемся добавить новый файл, названный path. Функция обратного
 * вызова может сохранить маркер для этого нового файла в **file_baton:
 * то значение, которое там сохраняется, должно быть передано
 * apply_textdelta.
 */
svn_error_t *(*add_file)(const char *path,
                        void *parent_baton,
                        apr_pool_t *file_pool,
                        void **file_baton);
/** Мы собираемся внести изменения в файл, названный path, который
 * принадлежит каталогу, идентифицируемому маркером parent_baton.
 *
 * Функция обратного вызова может сохранить маркер для этого нового файла
 * в **file_baton:
 * то значение, которое там сохраняется, должно быть передано
 * apply_textdelta.
 */
svn_error_t *(*open_file)(const char *path,
                        void *parent_baton,
                        apr_pool_t *file_pool,
                        void **file_baton);
/** Использование текстовых изменений, принимая новую редакцию файла.
```

```
/*
 * Маркер file_baton указывает на файл, который был создан или обновлен,
 * и на предшествующий файл, на основе которого он был создан; это
 * маркер, установленный одной из ранее вызванных функций обратного
 * вызова add_file или open_file.
 */
* Функция обратного вызова должна установить *handle для обработчика
* окна измененного текста: затем мы должны вызвать *handle для обработки
* успешно полученного окна с измененным текстом. Функция обратного
* вызова должна установить маркер *handler_baton на значение, которое мы
* должны передать в качестве аргумента маркера *handler.
*/
svn_error_t *(*apply_textdelta)(void *file_baton,
                               apr_pool_t *pool,
                               svn_txdelta_window_handler_t *handler,
                               void **handler_baton);

/** Мы завершили обработку файла, имеющего маркер file_baton
 * (установленный add_file или open_file). Мы больше не
 * собираемся использовать этот маркер, поэтому ресурсы, на которые он
 * ссылается, теперь могут быть освобождены.
 */
svn_error_t *(*close_file)(void *file_baton,
                           apr_pool_t *pool);

/** Обработка изменений завершена. Для полного выхода нужно сделать
 * следующий вызов с маркером edit_baton.
 */
svn_error_t *(*close_edit)(void *edit_baton,
                           apr_pool_t *pool);

/** Драйвер редактора решил высвободиться. Нужно разрешить редактору
 * изящно все подчистить, если в этом есть необходимость.
 */
svn_error_t *(*abort_edit)(void *edit_baton,
                           apr_pool_t *pool);
} svn_delta_editor_t;
```

Но разве это искусство?

Не могу сказать, что красота этого интерфейса сразу бросилась мне в глаза. Я не уверен, что и для Джима она была очевидной; может быть, он просто пытался побыстрее выставить нас с Беном за порог своего дома. Но он тоже довольно долго размышлял над этой проблемой и для определения поведения древовидной структуры подключил всю свою интуицию.

Первое, что поражает в дельта-редакторе, — это то, что он *выбирает* ограничения. Даже при отсутствии философских требований на редактирование дерева именно в глубину (или вообще в каком-нибудь другом порядке) интерфейс обязан в любом случае работать с деревом только в глубину, посредством взаимоотношения маркеров. Это делает использование интерфейса и его поведение более предсказуемым.

Второе удивительное свойство заключается в том, что, опять-таки благодаря маркерам, вся операция редактирования ненавязчиво переносит с собой все свое окружение. Маркер файла может содержать указатель на маркер его родительского

каталога, а маркер каталога может содержать указатель на маркер *его* родительского каталога (включая пустой родительский каталог при редактировании корневого каталога), и любой маркер может содержать указатель на глобальный маркер редактирования. Хотя индивидуальный маркер может быть одноразовым объектом — например, когда файл закрыт, его маркер удаляется, — любой другой маркер позволяет получить доступ к глобальному редактируемому окружению, которое может содержать, к примеру, номер версии, редактируемой на стороне клиента.

Таким образом, маркеры перезагружаются: в них предусмотрена область действия (то есть время существования, поскольку маркер сохраняется только на время существования пула, в котором он размещается) редактируемых частей, но в них также переносится и глобальное окружение.

Третье важное свойство заключается в том, что интерфейс определяет четкие границы между различными подоперациями, задействованными при выражении изменений дерева. Например, открытие файла всего лишь показывает, что от дерева к дереву в этом файле произошли какие-то изменения, но их детали не предоставляются; детализация дается при вызове функции `apply_textdelta`, но если этого не требуется, то вызывать эту функцию не обязательно. Точно так же открытие каталога показывает, что произошли изменения в нем или под ним, но если ничего другого и не требуется, можно просто закрыть каталог и продолжить работу. Эти разграничения являются следствием ориентированности интерфейса на *потоковую передачу данных*, выраженную в его вводном комментарии: «... вместо полного представления дельты дерева, мы определяем для потребителя стандартный способ обработки каждой порции дельты дерева по мере ее создания производителем». Может появиться соблазн разбивать на потоки только самые большие порции данных (то есть изменившиеся фрагменты файлов), но интерфейс дельта-редактора не поступается принципом и разбивает на потоки все изменения дерева, давая, таким образом, и производителю и потребителю четко распределенное управление пользованием пространства памяти, возможность получения индикации процесса и его непрерывность. Ценность этих свойств начала проявляться только после того, как мы испытали новый дельта-редактор в решении различных проблем. К примеру, нам захотелось осуществить резюмирование изменений, то есть найти способ отображения краткого обзора различий между двумя деревьями без их детализации. Это могло бы кому-нибудь пригодиться при определении, какие файлы рабочей копии подверглись в хранилище изменениям со времени последней проверки, но знать, что именно в них изменилось, при этом не нужно.

Вот как выглядит немного упрощенная версия того, как это работает: клиент сообщает серверу, на дереве какой версии базируется его рабочая копия, а затем сервер, используя дельта-редактор, сообщает клиенту, в чем заключается разница между деревом этой версии и деревом самой последней версии. В данном случае сервер является производителем, а клиент потребителем.

На примере ранее использованной в этой главе модели хранилища, в которую мы для создания версии 2 постепенно вносили изменения `/A/fish/tuna`, посмотрим, как это будет выглядеть в виде серии вызовов редактора, посланных сервером клиенту, чье дерево до сих пор соответствует версии 1. Решение о том,

что будет сообщаться о редактировании, абсолютно все сведения или только резюме, должно быть принято где-то при прохождении двух третей содержимого блока if:

```
svn_delta_editor_t *editor
void *edit_baton;

/* На самом деле это, конечно, должен быть передаваемый параметр. */
int summarize_only = TRUE;

/* На самом деле эти переменные должны быть объявлены в подпрограммах,
чтобы время их существования было ограничено рамками стека, подобно
тому как объекты, на которые они указывают, ограничены рамками
редактирования дерева.*/
void *root_baton;
void *dir_baton;
void *subdir_baton;
void *file_baton;

/* По тем же соображениям это должны быть подпулы, не стоящие рядом с пулом
верхнего уровня.*/
apr_pool_t *pool = svn_pool_create( );

/* Каждое использование интерфейса дельта-редактора начинается с запроса
конкретного редактора, совершающего необходимые действия,
например, потоковую передачу результатов редактирования по сети, чтобы
применить их к рабочей копии, и т. д.*/
Get_Update_Editor(&editor, &eb,
    some_repository,
    1, /* номер версии источника */
    2, /* номер версии приемника */
    pool);

/* Теперь мы управляем действиями редактора. В действительности эта
последовательность вызовов будет сгенерирована динамически тем кодом,
который пройдет по двум деревьям хранилища и осуществит
соответствующий вызов editor->foo( ). */

editor->open_root(edit_baton, pool, &root_baton);
editor->open_directory("A", root_baton, pool, &dir_baton);
editor->open_directory("A/fish", dir_baton, pool, &subdir_baton);
editor->open_file("A/fish/tuna", subdir_baton, pool, &file_baton);

if (!summarize_only)
{
    svn_txdelta_window_handler_t window_handler;
    void *window_handler_baton;
    svn_txdelta_window_t *window;

    editor->apply_textdelta(file_baton, pool
        apr_pool_t *pool,
        &window_handler,
        &window_handler_baton);
    do {
        window = Get_Next_TextDelta_Window(...);
        window_handler(window, window_handler_baton);
    }
}
```

```

    } while (window);
}

editor->close_file(file_baton, pool);
editor->close_directory(subdir_baton, pool);
editor->close_directory(dir_baton, pool);
editor->close_directory(root_baton, pool);
editor->close_edit(edit_baton, pool);

```

Как показано в этом примере, различие между краткими сведениями и полной версией изменений вполне естественным образом укладывается в границы интерфейса дельта-редактора, позволяя нам использовать для выполнения обеих задач один и тот же проход кода. Так уж случилось, что деревья двух версий расположены рядом (версия 1 и версия 2), но это совсем не обязательно. Тот же самый метод работал бы для любых двух деревьев, даже если их разделяло бы множество версий, что случается, когда рабочая копия долго не обновлялась. Он также будет работать и при обратном расположении двух деревьев — то есть когда более новая версия будет идти первой. Этим можно будет воспользоваться для отката изменений.

Абстракция как зрелищный спорт

Один из следующих показателей гибкости дельта-редактора проявляется, когда возникает потребность совершать над одним и тем же редактируемым деревом два или более различных действий. Одна из первых подобных ситуаций была связана с необходимостью обработки отмен. Когда пользователь прерывает обновление, обработчик сигнала перехватывает запрос и устанавливает флаг; затем, в течение работы, мы проверяем из различных мест этот флаг и, если он установлен, осуществляем аккуратный выход. Оказывается, в большинстве случаев наиболее безопасным местом для выхода является следующая граница входа или выхода из запроса функций редактора. Это было вполне банальной истиной для операций, не совершающих ввод/вывод на стороне клиента (типа резюмирования изменений и отличий), но это также было справедливо для многих операций, затрагивающих работу с файлами. В конечном итоге большинство работ по обновлению представляют собой простую внешнюю запись данных, и даже если пользователь целиком прерывает операцию по обновлению, обычно все же имеет смысл либо завершить запись, либо, при обнаружении прерывания, аккуратно отменить работу с любым из обрабатываемых файлов.

Но где же тогда проверять состояние флага? Мы можем жестко запрограммировать эти проверки в дельта-редакторе, чтобы одна из них возвращалась (по требованию) функцией `Get_Update_Editor()`. Но по всей видимости, это весьма слабое решение: дельта-редактор является библиотечной функцией, которая может быть вызвана из кода, которому требуется совершенно другая техника проверки отмен или вообще ничего не нужно.

Чуть лучшим может быть решение о передаче обратного вызова для проверки отмены и связанного с редактированием маркера функции `Get_Update_Editor()`. Возвращенный редактор будет периодически осуществлять обратный вызов на основе маркера и, в зависимости от возвращаемого значения, либо

продолжать нормальную работу, либо осуществлять возврат до ее окончания (если обратный вызов вернул null, то редактор вызываться не будет). Но эта схема также далека от идеала. Получается, что проверка на отмену — второстепенная задача: может быть, она понадобится при обновлении, а может быть, и не понадобится, но в любом случае тому способу, который используется в процессе обновления, она ничего не дает. В идеале не стоит связывать эти два процесса в одном программном коде, тем более, что мы пришли к заключению, что в большинстве случаев, операции не нуждаются в уточненном управлении проверкой отмены, так или иначе, система разграничений вызова редактора сама бы неплохо со всем спривилась.

Отмена — лишь один из примеров сопутствующих задач, связанных с редактированием изменений дерева. Мы столкнулись, или думали, что столкнулись, с похожими проблемами в отслеживании подтвержденных мест назначения при передаче изменений от клиента на сервер, в сообщении об обновлении или передаче сообщения о ходе процесса пользователю и в различных других ситуациях. Естественно, мы искали способ абстрагироваться от этих дополнительных нагрузок, чтобы не загромождать ими основной код. На самом деле мы настолько усердно его искали, что уже изначально переабстрагировались:

```
/** Компоновка редактора editor_1 и его маркера с редактором editor_2 и его
 * маркером.
 *
 * Возвращение нового редактора в new_editor (размещаемый в пуле), в котором
 * каждая функция fun вызывает editor_1->fun, а затем editor_2->fun,
 * с соответствующими маркерами.
 *
 * Если editor_1->fun возвращает ошибку, эта ошибка возвращается из
 * new_editor->fun, и editor_2->fun уже не вызывается: в противном случае
 * значение, возвращенное new_editor->fun, аналогично значению, возвращенному
 * editor_2->fun.
 *
 * Если функция редактора возвращает null, то вызова не происходит, и это
 * не считается ошибкой.
 */
void
svn_delta_compose_editors(const svn_delta_editor_t **new_editor,
                           void **new_edit_baton,
                           const svn_delta_editor_t *editor_1,
                           void *edit_baton_1,
                           const svn_delta_editor_t *editor_2,
                           void *edit_baton_2,
                           apr_pool_t *pool);
```

Хотя, как оказалось, на этом дело не кончилось, и мы скоро увидим, почему. Я по-прежнему считаю это свидетельством совершенства интерфейса редактора. Составные редакторы вели себя предсказуемо, сохраняя чистоту кода (поскольку никакая отдельная функция редактора не должна была заботиться о деталях, вызванных параллельным редактором до или после него), и они прошли тест на сочетаемость: вы можете взять редактор, который сам был результатом композиции, и скомпоновать его с другими редакторами, и все это должно просто работать. Это работало, поскольку все редакторы были согласованы

по основной форме выполняемой ими операции, даже при том что они могли делать с данными совершенно разные вещи.

Вы, конечно, можете сказать, что для пущей элегантности я до сих пор не обратил внимания на структуру редактора. Но в конечном счете она была более абстрактной, чем нам требовалось. Большую часть функциональности, изначально обеспеченнной использованием составных редакторов, мы позже переписали под использование клиентских обратных вызовов, передаваемых процедурам создания редакторов. Хотя сопутствующие действия обычно совпадали с границами вызова редактора, они зачастую не были свойственны *всем* границам вызова или даже большинству из них. В результате получалось слишком высокое соотношение инфраструктуры к работе: установкой полноценного параллельного редактора мы невольно наводили читателей кода на мысль, что сопутствующие действия будут вызываться чаще, чем это было на самом деле.

После того как мы сделали все, что могли, со структурой редактора и отступили, у нас все еще оставалась возможность собственноручно отменить все это в случае необходимости. Сегодня в Subversion отмена производится с использованием структуры, созданной вручную. Конструктором редактора проверки отмены в качестве параметра задействуется другой редактор, занимающийся основными операциями:

```
/** Установка *editor и *edit_baton в редакторе отмены, в котором
 * помещаются wrapped_editor и wrapped_baton.
 */
* При вызове каждой из его функций редактор вызовет cancel_func с маркером
* cancel_baton и продолжит вызов соответствующей вовлеченнной функции, если
* cancel_func вернет SVN_NO_ERROR.
*/
* Если cancel_func возвращает значение NULL, в качестве значения *editor
* устанавливается wrapped_editor, а в качестве значения *edit_baton
* устанавливается wrapped_baton.
*/
svn_error_t *
svn_delta_get_cancellation_editor(svn_cancel_func_t cancel_func,
                                   void *cancel_baton,
                                   const svn_delta_editor_t *wrapped_editor,
                                   void *wrapped_baton,
                                   const svn_delta_editor_t **editor,
                                   void **edit_baton,
                                   apr_pool_t *pool);
```

Мы также осуществляли несколько условных отладочных трассировок, используя схожий процесс собственноручного создания структуры. Другие сопутствующие задачи — главным образом индикация прогресса, сообщения о происходящем и подсчет мест назначения — осуществляются посредством обратных вызовов, передаваемых конструктором редактора и (если указатель не пустой) вызываемых редактором в нескольких местах, где в них есть необходимость.

Интерфейс редактора продолжает обеспечивать сильное объединяющее влияние по всему коду Subversion.

Возможно, это покажется странным — хвалить программный интерфейс (API), который сначала соблазняет своих пользователей уйти в сверхабстракцию. Но это искушение по большому счету было лишь побочным эффектом

полноценного удовлетворения требований проблемы потоковой передачи данных об изменениях дерева — благодаря этому проблема стала выглядеть настолько податливой, что нам захотелось, чтобы все другие проблемы превратились в эту проблему! Когда подгонка не удавалась, мы отступали, но конструкторы редактора по-прежнему предоставляли каноническое место для вставки обратных вызовов, и внутренние границы действия редакторов помогали размышлять о том, когда осуществлять эти обратные вызовы.

Выводы

Реальная сила этого API и, я подозреваю, любого хорошего API состоит в том, что он формирует направление чьих-то рассуждений. Порядок всех операций, задействованных в Subversion при модификации дерева, примерно одинаков. Это не только существенно экономит время, которое новички вынуждены затрачивать на изучение существующего кода, но и дает новому коду ясную модель, которой можно следовать, и разработчики не преминули воспользоваться этой подсказкой. Например, функция *svn sync*, которая зеркально отображает одни действия с хранилищем на другом хранилище и была добавлена в Subversion в 2006 году, через шесть лет после появления дельта-редактора использует его интерфейс для передачи всех действий. Разработчик функции не только сэкономил на необходимости создания механизма передачи изменений, но даже на том, что ему вообще не понадобилось *рассматривать вопрос создания* подобного механизма. А те, кто теперь разбирается с новым кодом, находят, что он во многом похож на то, что они видели, когда разбирались с ним в первый раз.

Здесь мы имеем дело с существенными выгодами. Наличие надлежащего API не только сокращает время изучения, оно также избавляет сообщество разработчиков от необходимости проводить какие-то проектные дискуссии, которые выливаются в нескончаемую переписку. Возможно, это нельзя сравнять с чисто технической или эстетической красотой, но в проекте с многочисленными участниками и круговоротом изменений это та самая красота, которой можно воспользоваться.

3 Самый красивый код, который я никогда не писал

Джон Бентли (*Jon Bentley*)

Однажды в адрес одного великого программиста я услышал такую похвалу: «Удаляя код, он делает его более функциональным». Французский писатель и авиатор Антуан де Сент-Экзюпери выразил ту же самую мысль в обобщенном виде, сказав: «Конструктор понимает, что достиг совершенства не в тот момент, когда к конструкции уже нечего добавить, а когда из нее уже нечего убрать». В программной сфере самый красивый код, превосходные функции и идеальные программы порой представляются вообще чем-то неопределенным.

А вести речь о неопределенных вещах, конечно же, трудно. В данной главе я попробую справиться с этой непростой задачей, представив новый анализ рабочего цикла классической программы быстрой сортировки — Quicksort. В первом разделе я подготовлю почву для критического анализа, рассматривая Quicksort с собственной точки зрения. Следующий раздел является основой главы. Мы начнем с добавления к программе еще одного счетчика, затем займемся переделкой кода, последовательно добиваясь уменьшения его размера и увеличения эффективности, пока лишь несколько строк кода не дадут нам картину его среднего времени исполнения. В третьем разделе суммируются методики и представляется несколько сокращенный анализ значимости деревьев двоичного поиска. В двух последних разделах выражается самая суть главы, призванная помочь вам в создании более качественных программ.

Мой самый красивый код

Когда Грег Уилсон (Greg Wilson) впервые объяснил мне замысел этой книги, я подумал: а каким был мой самый красивый код? После того как этот вопрос не давал мне покоя чуть ли не весь день, я понял, что ответ довольно прост — это быстрая сортировка Quicksort. К сожалению, на один вопрос есть три разных ответа, зависящих от того, как он конкретно сформулирован.

Я написал тезисы по алгоритмам декомпозиции и пришел к заключению, что алгоритм Quicksort, созданный Чарльзом Энтони Ричардом Хоаром (C.A.R. Hoare, «Quicksort», Computer Journal, 5), является для всех них бесспорным дедушкой. Красивый алгоритм решения фундаментальной проблемы, который может быть воплощен в превосходном программном коде.

Мне нравился алгоритм, но я всегда крайне настороженно относился к самому глубокому из его циклов. Однажды я провел два дня за отладкой довольно сложной программы, основанной на этом цикле, а затем годами бережно копировал этот код в те программы, где требовалось выполнение сходной задачи. Цикл позволял решать мои проблемы, но *по-настоящему* я так и не понимал его работу.

Впоследствии я изучил тонкую схему разбиения, придуманную Нико Ломуто (Nico Lomuto), и был, наконец, в состоянии написать быструю сортировку, работу которой мог понять и даже утверждать, что все работает правильно. Замечание Уильяма Странка-младшего (William Strunk Jr.), что «доходчивое письмо отличается краткостью» применимо к коду точно так же, как и к английскому языку, поэтому я последовал его наставлению «избегать лишних слов» (совет из книги «Элементы стиля»). В конечном итоге я сократил код примерно с сорока до почти что двенадцати строк. Поэтому если сформулировать вопрос следующим образом: «Каким был самый красивый небольшой фрагмент кода из всех, написанных вами?», то я отвечу, что это функция Quicksort из моей книги «Programming Pearls», второе издание (Addison-Wesley). Эта функция, написанная на языке Си, показана в примере 3.1. В следующем разделе мы изучим и улучшим этот пример.

Пример 3.1. Функция быстрой сортировки

```
void quicksort(int l, int u)
{
    int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

Этот код, будучи вызванным с параметрами `quicksort(0, n-1)`, осуществляет сортировку глобального массива `x[n]`. Используемые функцией, два параметра являются индексами подмассива, подлежащего сортировке: `l` — нижний, а `u` — верхний индекс. Вызов функции `swap(i, j)` обменивает содержимое элементов `x[i]` и `x[j]`. Первая перестановка осуществляет произвольный выбор элемента разделения, с равной вероятностью выбирая целое число в диапазоне между `l` и `u`.

В книге «Programming Pearls» содержится подробный разбор и доказательство работоспособности функции быстрой сортировки. Далее в этой главе я буду

считать, что читатель знаком с Quicksort на уровне, представленном в этом описании и в самых простых учебниках по составлению алгоритмов.

Если изменить постановку вопроса и спросить: «Каким был ваш самый красивый из наиболее востребованных фрагментов кода?», мой ответ не изменится — Quicksort. В статье, написанной совместно с М. Д. Макилроем (M. D. McIlroy, «Engineering a sort function», Software—Practice and Experience, Vol. 23, № 11), описана серьезная ошибка производительности в почтенной Unix-функции `qsort`. Мы намеревались создать для библиотеки Си новую функцию `sort` и рассмотрели множество различных алгоритмов для решения этой задачи, включая сортировку слиянием (*Merge Sort*) и пирамidalную сортировку (*Heap Sort*). После сравнения нескольких возможных вариантов создания функции мы остановились на версии алгоритма Quicksort. Здесь описывается, как мы создавали новую функцию, которая была понятнее, быстрее и проще своих конкурентов — в частности, за счет меньшего размера. Высказывание мудреца Гордона Белла (Gordon Bell): «Самые дешевые, быстрые и наиболее надежные компоненты компьютерной системы в ней просто не установлены», оказалось истиной. Эта функция не отличалась широкой востребованностью в течение более чем десяти лет, и это при отсутствии каких-либо сообщений об ошибках.

Рассматривая преимущества, которые могут быть получены за счет сокращения размеров кода, я в конце концов задал себе третий вариант вопроса, с которого начиналась эта глава: «Что представляет из себя самый лучший код, который ты *никогда не писал?*». Как из меньшего я мог получить нечто более значительное? Вопрос опять-таки относился к Quicksort, а именно к анализу ее производительности. Эта история рассказана в следующем разделе.

Усиление отдачи при сокращении размеров

Quicksort — очень элегантный алгоритм, сам собой напрашивавшийся на тонкий анализ. Где-то в 1980 году я провел замечательную дискуссию с Тони Хоаром (Tony Hoare) об истории его алгоритма. Он рассказал мне, что когда он только разработал Quicksort, то думал, что пустить его в обращение будет совсем нетрудно, и ограничился написанием своей классической статьи «*Quicksort*» после того, как получил возможность проанализировать ожидаемое время выполнения.

Нетрудно понять, что в худшем случае Quicksort потратит на сортировку массива из n элементов примерно n^2 интервалов времени. В лучшем случае в качестве разделяющего элемента будет выбрано среднее значение, в силу чего при сортировке массива будет сделано около $n \lg n$ сравнений. Итак, сколько же в среднем будет использовано операций сравнения для массива случайно выбранных значений, состоящего из n различных элементов?

Проведенный Хоаром анализ этого вопроса не лишен красоты, но, к сожалению, превосходит математические способности многих программистов. Когда я преподавал студентам Quicksort, мне приходилось расстраиваться, что многие из них, как ни старались, просто не понимали моих доказательств. Теперь мы

предпримем атаку на эту проблему экспериментальным путем. Начнем с программы, созданной Хоаром, и, в конечном счете, приблизимся к проделанному им анализу.

Наша задача состоит в модификации кода произвольной сортировки Quicksort, приведенного в примере 3.1, в целях анализа среднего числа сравнений, использованных для сортировки массива, состоящего из отличающихся друг от друга исходных значений. Мы попытаемся также максимально во всем разобратьсяся, используя минимум кода, времени на выполнение и пространства.

Чтобы определить среднее число сравнений, сначала мы добавим к программе их подсчет. Для этого перед тем как производить сравнение во внутреннем цикле (пример 3.2), придадим приращение переменной comps.

Пример 3.2. Quicksort, приспособленный для подсчета сравнений во внутреннем цикле

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

Запустив программу для одного значения n , мы увидим, сколько сравнений будет проделано именно при этом запуске. Если мы запустим программу многократно, для множества значений n , и проведем статистический анализ результатов, то сможем заметить, что в среднем Quicksort проводит около $1,4 n \lg n$ сравнений для сортировки n элементов.

Это неплохой способ получения сведений о поведении программы. Тринадцать строк кода и несколько экспериментов могут выявить многое. В известной цитате, авторство которой приписывается таким авторам, как Блез Паскаль (Blaise Pascal) и Т. С. Элиот (T. S. Eliot), утверждается, что «если бы у меня было больше времени, я написал бы письмо покороче». У нас есть время, поэтому давайте поэкспериментируем с кодом и попытаемся создать более короткую (и лучшую) программу.

Поиграем в ускорение эксперимента, пытаясь улучшить статистическую точность и понимание всего происходящего в программе. Поскольку внутренний цикл всегда совершает строго $u-l$ сравнений, мы можем несколько ускорить работу программы, вычисляя количество этих сравнений в единственной операции за пределами цикла. В результате этого изменения получится Quicksort, показанный в примере 3.3.

Пример 3.3. Внутренний цикл Quicksort, с приращаемой переменной, вынесенной за пределы цикла

```
comps += u-l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

Эта программа сортирует массив и подсчитывает количество сравнений, использованных при этом. Однако если нашей целью является лишь подсчет сравнений, сама сортировка не нужна. В примере 3.4 удалена «реальная работа»

по сортировке элементов, и сохранена только «схема» различных вызовов, сделанных программой.

Пример 3.4. Схема Quicksort, сокращенная до подсчета

```
void quickcount(int l, int u)
{
    int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

Эта программа работает, благодаря «произвольному» способу, которым Quicksort выбирает разделяющий элемент, и благодаря предположению, что значения всех элементов отличаются друг от друга. Теперь время работы новой программы пропорционально n , а требуемое пространство сокращено на размеры стека рекурсии, который в среднем пропорционален $\lg n$, тогда как код примера 3.3 требует пространства памяти, пропорционального n .

Тогда как в настоящей программе индексы массива (l and u) играют весьма важную роль, в версии схемы они не имеют никакого значения. Мы можем заменить оба этих индекса одним целым значением (n), которое определяет размер сортируемого подмассива (пример 3.5).

Пример 3.5. Схема Quicksort с единственным аргументом размера

```
void qc(int n)
{
    int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

Теперь будет более естественным поменять название этой функции на *счетчик сравнений*, возвращающий их количество, затрачиваемое на один произвольный запуск Quicksort. Эта функция показана в примере 3.6.

Пример 3.6. Схема Quicksort, реализованная в виде функции

```
int cc(int n)
{
    int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

Во всех примерах – 3.4, 3.5 и 3.6 – решается одна и та же основная задача, и они работают с одинаковыми затратами времени и пространства используемой памяти. В каждом последовательном варианте форма функции совершенствуется, в результате чего она становится понятнее и короче своей предшественницы.

В определении парадокса изобретателя Дьюрдь Пойя (George Polya) сказал, что «наиболее амбициозный план может иметь больше шансов на успех». Теперь мы попытаемся воспользоваться этим парадоксом при анализе Quicksort. До сих пор нами ставился вопрос: «Сколько сравнений делает Quicksort, будучи однократно запущенным с массивом из n элементов?». Теперь зададим более амбициозный вопрос: «Сколько сравнений в среднем делает Quicksort для произвольного массива из n элементов?». Мы можем расширить пример 3.6, чтобы привести его к псевдокоду, показанному в примере 3.7.

Пример 3.7. Псевдокод для определения среднего количества сравнений, проводимых Quicksort

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

Если на входе имеется всего один элемент, Quicksort не использует ни одного сравнения, как и в примере 3.6. Для большего количества n этот код рассматривает каждое разделяющее значение m (с равной вероятностью лежащего в диапазоне от первого элемента до последнего) и определяет, во что обходится разделение в каждом конкретном месте. Затем код подсчитывает сумму этих значений (в связи с чем рекурсивно решает одну задачу с размером $m-1$, а другую — с размером $n-m$), а затем делит эту сумму на n , чтобы вернуть среднее значение.

Если бы мы смогли вычислить это количество, наши эксперименты стали бы намного ценнее. Вместо того чтобы оценивать среднее значение путем проведения множества экспериментов, используя единственное значение n , мы получим правильное среднее значение в результате проведения всего лишь одного эксперимента. К сожалению, эта ценность дорого обходится: время выполнения программы пропорционально 3^n (было бы интересно самостоятельно поупражняться в анализе затрачиваемого времени, используя технику, описанную в этой главе). Код, приведенный в примере 3.7, затрачивает столь значительное время на исполнение, поскольку он по многу раз вычисляет промежуточные ответы. Если программа делает что-либо подобное, мы можем в большинстве случаев использовать для хранения промежуточных ответов *динамическое программирование*, чтобы избежать их повторных вычислений. В данном случае мы введем в код таблицу $t[N+1]$, в которой $t[n]$ хранит $c(n)$ и вычисляет значения этого выражения в нарастающем порядке. Переменной N мы позволим обозначать максимальный размер n , представляющий собой размер сортируемого массива. Результат показан в примере 3.8.

Пример 3.8. Подсчеты для Quicksort с помощью динамического программирования

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
```

```
sum += n-1 + t[i-1] + t[n-i]
t[n] = sum/n
```

Эта программа является приблизительной копией примера 3.7, в которой $c(n)$ заменена на $t[n]$. Время ее выполнения пропорционально N^2 , а требуемое пространство памяти пропорционально N . Одно из ее преимуществ состоит в том, что по окончании выполнения массив t содержит действительное среднее значение (а не только подсчет средних примера) для массива элементов от 0 до N . Эти значения могут быть проанализированы для того, чтобы привести к пониманию функциональной формы ожидаемого числа сравнений, используемых Quicksort.

Теперь мы можем провести дальнейшее упрощение программы. Сначала выведем элемент $n-1$ за пределы цикла, как показано в примере 3.9.

Пример 3.9. Подсчеты для Quicksort с кодом, выведенным за пределы цикла

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 + sum/n
```

Теперь проведем дальнейшую настройку цикла, воспользовавшись симметричностью. Когда, к примеру, n равно 4, внутренний цикл вычисляет следующую сумму:

```
t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]
```

В последовательности пар индекс первого элемента возрастает, а второго – убывает. Поэтому мы можем переписать сумму следующим образом:

```
2 * (t[0] + t[1] + t[2] + t[3])
```

Этой симметричностью можно воспользоваться, чтобы получить Quicksort, показанный в примере 3.10.

Пример 3.10. Подсчеты для Quicksort с использованием симметричности

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

Однако этот код опять-таки нерационален, поскольку он снова и снова вычисляет одну и ту же сумму. Вместо добавления всех предыдущих элементов мы можем задать начальные условия sum за пределами цикла и добавить следующие элементы, чтобы получить код, показанный в примере 3.11.

Пример 3.11. Подсчеты для Quicksort с удаленным внутренним циклом

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

Эта небольшая программа действительно может быть полезна. За время, пропорциональное N , она выдает таблицу реально ожидаемых времен прогона Quicksort для каждого целого числа от 1 до N .

Код примера 3.11 можно напрямую применить в электронной таблице, где значения тут же становятся доступными для последующего анализа. В табл. 3.1 показаны ее первые строки.

Таблица 3.1. Вывод табличной реализации примера 3.11

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2,667
4	7,333	4,833
5	17	7,4
6	31,8	10,3
7	52,4	13,486
8	79,371	16,921

В этой таблице первая строка чисел получила начальные значения трех констант, взятых из кода. В системе записей электронной таблицы следующая строка чисел (третья строка таблицы) вычисляется с использованием следующих связей:

$$A3 = A2+1 \quad B3 = B2 + 2*C2 \quad C3 = A3-1 + B3/A3$$

Электронная таблица заполняется перетаскиванием этих (относительных) связей вниз. Эта электронная таблица – реальный претендент на «самый красивый из когда-либо написанных мною кодов», если использовать критерии достижения весьма существенного результата в нескольких строчках кода.

А если нам не понадобятся все эти значения? Если мы предпочтем подвергнуть анализу только некоторые из них (например все значения, кратные двум, от 2^0 до 2^{32})? Хотя в примере 3.11 выстраивается полная таблица t , используются только ее самые последние значения.

Поэтому мы можем заменить линейное пространство таблицы $t[]$ неизменным пространством переменной t , как показано в примере 3.12.

Пример 3.12. Подсчеты для Quicksort, заключительная версия

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

Затем мы можем вставить еще одну строку кода, чтобы проверить приемлемость n и распечатать именно те результаты, которые нам нужны.

Эта маленькая программа является завершающим шагом нашего долгого пути. Все пройденное в этой главе вполне вписывается в наблюдение Алана

Перлиса (Alan Perlis): «Простота не предшествует сложности, а напротив, следует за ней».

Перспективы

В табл. 3.2 сведены показатели программ, использованных в этой главе для анализа Quicksort.

Таблица 3.2. Развитие программ подсчета сравнений в Quicksort

Номер примера	Количество строк кода	Тип ответа	Количество ответов	Время выполнения	Пространство
2	13	Выборочный	1	$n \lg n$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg n$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Точный	"	3^n	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	Точный	N	N	1

Каждый отдельный шаг в эволюции нашего кода был предельно прост и понятен; самым тонким, пожалуй, был переход от выборочного ответа в примере 3.6 к точному ответу в примере 3.7. В процессе изменений, по мере того как код становился все более быстродействующим и полезным, он к тому же сокращался в размере. В середине XIX столетия Роберт Броунинг (Robert Browning) заметил, что «в меньшем заключается большее», и данная таблица помогает дать количественную оценку одного из примеров этой минималистской философии.

Мы рассмотрели три существенно различающихся типа программ. Примеры 3.2 и 3.3 представляли собой работоспособный Quicksort, приспособленный для подсчета сравнений в процессе сортировки реального массива. Примеры от 3.4 до 3.6 были реализацией простой модели Quicksort: они имитировали однократный запуск алгоритма без осуществления реальной работы по сортировке. Примеры от 3.7 до 3.12 явились воплощением более сложной модели: в них шел подсчет истинного среднего количества сравнений вообще без сортировки какого-либо конкретного запуска.

Из технических приемов, которые использовались при выполнении каждой программы, можно составить следующие группы:

- примеры 3.2, 3.4, 3.7: фундаментальное изменение формулировки проблемы;
- примеры 3.5, 3.6, 3.12: незначительное изменение определения функции;

- пример 3.8: новая структура данных для осуществления динамического программирования.

Это типовые приемы. Зачастую программу можно упростить, задавшись вопросом: «Какую именно проблему нужно решить?» или «Существует ли для решения этой проблемы более подходящая функция?».

Когда я показал эту аналитику студентам, программа окончательно уменьшилась до нуля строк и растворилась в клубах математического тумана. Пример 3.7 мы можем истолковать заново в виде следующего рекуррентного соотношения:

$$C_0 = 0 \quad C_n = (n - 1) + (1/n) \sum_{1 \leq i \leq n} C_{i-1} + C_{n-i}.$$

Точно такой же подход был применен Хоаром и позже представлен Д. Е. Кнутом (D. E. Knuth) в его классической книге «The Art of Computer Programming, Volume 3: Sorting and Searching» (Addison-Wesley). Программные трюки, позволившие выразить все это по-новому и воспользоваться симметрией, в результате чего появился код примера 3.10, дают возможность упростить рекурсивную часть до следующего вида:

$$C_n = n - 1 + (2/n) \sum_{0 \leq i \leq n-1} C_i.$$

Методика Кнута по удалению знака суммирования находит свое приблизительное отображение в примере 3.11, который может быть выражен по-новому в виде системы двух рекуррентных соотношений двух неизвестных:

$$C_0 = 0 \quad S_0 = 0 \quad S_n = S_{n-1} + 2C_{n-1}, \quad C_n = n - 1 + S_n / n.$$

Для получения следующего решения Кнут использует математическую методику «коэффициента суммирования»:

$$C_n = (n+1)(2H_{n+1} - 2) + 2n \approx 1,386n \lg n,$$

где H_n обозначает n -е гармоническое число $1 + 1/2 + 1/3 + \dots + 1/n$. Таким образом, мы плавно перешли от экспериментов с программами, подкрепленных исследованиями, к полностью математическому анализу их поведения.

Этой формулой мы завершаем наши поиски, следуя известному совету Эйнштейна: «Делайте все как можно проще, но не за счет упрощения».

Бонусный анализ

Известное высказывание Гёте гласит, что «архитектура — это застывшая музыка». Точно так же я могу утверждать, что «структуры данных — это застывшие алгоритмы». И если мы заморозим алгоритм Quicksort, то получим структуру данных дерева двоичного поиска. В публикации Кнута представляется эта структура, и анализируется время выполнения с рекуррентным соотношением, подобным тому, что имеет место в Quicksort.

Если мы хотели бы проанализировать среднюю цену вставки элемента в дерево двоичного поиска, то могли бы начать с кода, прибавить к нему подсчет сравнений, а затем провести эксперимент над накопленными данными. Затем

можно было бы упростить этот код (и сделать его эффективнее), используя способ, весьма близкий к примененному в предыдущем разделе. Более простым было бы решение определить новый Quicksort, использующий метод *идеального разбиения*, который оставляет элементы в том же самом относительном порядке с обеих сторон. Такой Quicksort является изоморфным к деревьям двоичного поиска, показанным на рис. 3.1.

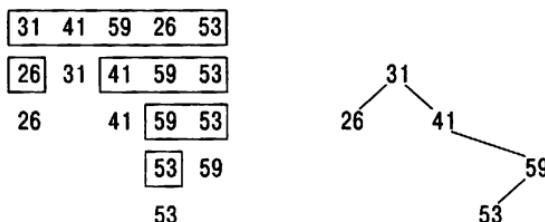


Рис. 3.1. Quicksort с идеальным разбиением и соответствующее дерево двоичного поиска

Расположенные слева прямоугольники отображают Quicksort с идеальным разбиением в процессе работы, а граф справа показывает соответствующее дерево двоичного поиска, которое было построено на тех же исходных данных. Эти два процесса используют не только одинаковое количество сравнений, они производят абсолютно одинаковую *последовательность*. Следовательно, наш предыдущий анализ средней производительности Quicksort при сортировке случайно выбранных значений отличающихся друг от друга элементов дает нам среднее количество сравнений, чтобы вставить произвольно переставленные отличающиеся друг от друга элементы в дерево двоичного поиска.

Что значит «написать»?

В некотором смысле я «написал» программы примеров 3.2–3.12. Сначала я написал их в виде набросков, затем на классной доске перед студенческой аудиторией и, наконец, в данной главе. Я систематически обращался к программам, уделил много времени их анализу и уверен в их правильности. Хотя, кроме создания электронной таблицы в примере 3.11, я никогда не запускал ни один из этих примеров в качестве компьютерной программы. Почти за два десятилетия, проведенных в Bell Labs, я учился у многих (и особенно у Брайена Кернигана, чей урок по программированию представлен в главе 1 этой книги), что «написание» программы для публичной демонстрации — это нечто большее, чем простой набор символов. Кто-то воплощает программу в коде, сначала тестирует ее несколькими запусками, затем основательно обустраивает ее саму, необходимые для ее работы драйверы, составляет библиотеку примеров и систематически над всем этим работает. В идеале проверенный компиляцией исходный код включается в текст автоматически, без чьего-либо вмешательства.

Я написал код примера 3.1 (и весь код в книге «Programming Pearls») именно с такими твердыми понятиями.

Из соображений чести я не хотел, чтобы моя репутация пострадала из-за того, что я никогда не пробовал в работе примеры 3.2–3.12. Почти сорок лет программистского стажа привили мне глубокое уважение к трудностям ремесла (а если быть точнее, то презренный страх перед ошибками). Я пошел на компромисс и выполнил пример 3.11 в электронной таблице, к которой добавил дополнительный столбец, придающий решению законченную форму. Вообразите мое восхищение (и облегчение), когда я получил полное соответствие этих двух форм! В итоге я предлагаю миру эти красивые, ненаписанные программы с определенной уверенностью в их корректности, тем не менее мучительно осознавая возможность существования невыявленной ошибки. Я надеюсь, что те глубокие достоинства, которые я в них обнаружил, не будут заслонены неизначительными дефектами.

В том дискомфорте, который возник от представления этих ненаписанных программ, меня утешает проницательность Алана Перлиса (Alan Perlis), который сказал: «Может быть, программное обеспечение не похоже ни на что другое, предназначеннное для отбраковки: что вся суть состоит в том, чтобы рассматривать его как некий мыльный пузырь?».

Вывод

У красоты есть множество источников. В этой главе мы сконцентрировались на той красоте, которая возникает из простоты, изящества и краткости. Эта всеобъемлющая тема нашла свое выражение в следующих афоризмах.

- Страйтесь наращивать функциональность, удаляя код.
- Конструктор понимает, что достиг совершенства не в тот момент, когда к конструкции уже нечего добавить, а когда из нее уже нечего убрать (Сент-Экзюпери).
- В программном обеспечении самый красивый код, самые красивые функции и самые красивые программы иногда вообще не встречаются.
- Живое письмо отличается краткостью. Уберите пустые слова (Странк и Уайт).
- Самые дешевые, быстродействующие и надежные компоненты компьютерной системы в ней не установлены (Белл).
- Страйтесь усилить отдачу, сократив размеры.
- Если бы у меня было больше времени, я написал бы письмо покороче (Паскаль).
- Парадокс изобретателя: Наиболее амбициозный план может иметь больше шансов на успех (Пойя).
- Простота не предшествует сложности, а напротив, следует за ней (Перлис).
- В меньшем заключается большее (Броунинг).
- Делайте все как можно проще, но не за счет упрощения (Эйнштейн).

- Программное обеспечение иногда нужно рассматривать как мыльный пузырь (Перлис).
- Ищите красоту в простоте.

На этом урок заканчивается. Действуйте по образцу и подобию.

Для тех, кто хочет получить более определенные советы, здесь приводятся некоторые идеи, сгруппированные в три основные категории.

Анализ программ

Одним из способов получения сведений о поведении программы является ее оснащение инструментальными средствами и запуск с эталонными данными, как в примере 3.2. Тем не менее зачастую мы заинтересованы не столько всей программой, сколько ее отдельными аспектами. В таком случае мы рассматриваем, к примеру, только среднее число сравнений, использованных в Quicksort, и игнорируем многие другие аспекты. Сэджвик (Sedgewick) изучал такие вопросы, как требуемое пространство памяти и многие другие компоненты времени выполнения для различных вариантов Quicksort. Сконцентрировавшись на ключевых вопросах, мы можем игнорировать (на какое-то время) другие аспекты программы. В одной из моих статей описано, как однажды я столкнулся с проблемой эволюции производительности *полосовой эвристики* (strip heuristic) для определения примерного количества поездок, совершаемых коммивояжером через N пунктов в пределах площади распространения изделия. Я прикинул, что вся программа для решения этой задачи может занять 100 строк кода. После того как я предпринял ряд шагов в духе того, что мы рассматривали в данной главе, я воспользовался для достижения большей точности симуляцией, уместившейся в десятке строк (а после завершения разработки этого небольшого симулятора я обнаружил, что Бердвид (Beardwood) и другие выразили мою симуляцию по-другому, в виде двойного интеграла, и решили, таким образом, задачу математически двумя десятилетиями ранее).

Небольшие фрагменты кода

Я уверен, что компьютерное программирование является настоящим искусством, и я согласен с Пойя, что мы «приобретаем любой практический навык путем подражания и решения реальных задач». Поэтому программисты, стремящиеся создавать красивый код, должны читать красивые программы и подражать изученным техническим приемам при написании своих собственных программ. Я считаю, что одним из наиболее полезных объектов для наработки практики являются небольшие фрагменты кода, размером, скажем, в один-два десятка строк. Я очень упорно, но с большим вдохновением готовил второе издание книги «Programming Pearls», исполняя каждый фрагмент кода и работая над тем, чтобы все свести к его смыслу. Я надеюсь, что другие получали не меньшее удовольствие, читая код, чем получал я, когда его писал.

Программные системы

Следуя специфике, я скрупулезно описал решение одной маленькой задачи. Я уверен, что успех применения основных принципов распространяется и на более сложные задачи.

няется не на маленькие фрагменты кода, а скорее, идет дальше, в большие программные комплексы и огромные компьютерные системы. Парнас (Parnas) предлагает методики сведения системы к ее основам. Чтобы незамедлительно этим воспользоваться, не забудьте откровение Тома Даффа (Tom Duff): «Займствуйте код, где только можно».

Признательность

Я благодарен за весьма существенные замечания Дану Бентли (Dan Bentley), Брайену Кернигану (Brian Kernighan), Энди Ораму (Andy Oram) и Дэвиду Уэйссу (David Weiss).

4 Пойск вещей

Tim Брэй (*Tim Bray*)

Компьютеры могут вычислять, но зачастую люди используют их не для этого. Чаще всего компьютеры хранят и извлекают информацию. *Извлечение* подразумевает поиск, и со временем изобретения Интернета поиск стал доминирующей прикладной задачей, выполняемой людьми на компьютерах.

По мере разрастания объема данных — как в абсолютном исчислении, так и относительно количества людей, или компьютеров, или чего-нибудь еще, поиск фактически занимал все большее и большее место в жизни программистов. Лишь немногим приложениям не нужно выискивать подходящие фрагменты в каких-нибудь информационных хранилищах.

Тема поиска — одна из самых обширных в компьютерной науке, поэтому я не буду пытаться рассмотреть ее во всем многообразии или обсуждать механику поиска. Я лучше подробно рассмотрю только одну простую методику поиска и сфокусирую внимание на альтернативах выбора поисковых технологий, которые могут быть едва различимы.

Фактор времени

Действительно, нельзя говорить о поиске, не затрагивая фактора времени. Применительно к проблемам поиска есть два понятия времени. Первое — это время, затрачиваемое на ведение поиска, в течение которого пользователь находится в ожидании, уставившись на какое-нибудь сообщение, вроде «Идет загрузка...». А второе — это время, затрачиваемое программистом, создающим функцию поиска, проводимое его руководством и клиентами в ожидании ввода программы в эксплуатацию.

Проблема: Данные веб-блога

Чтобы почувствовать, как поиск работает в реальной жизни, рассмотрим типовую проблему. Имеется каталог, содержащий регистрационные файлы моего веб-блога (<http://www.tbray.org/ongoing>) с начала 2003 по конец 2006 года;

на время написания данной главы в них зафиксировано 140070104 транзакций, и под них занято 28489788532 байт (несжатой информации). Все эти статистические данные при хорошо организованном поиске могут ответить на массу вопросов об объеме трафика и круге читателей.

Сначала рассмотрим простой вопрос: какие статьи были прочитаны наибольшее количество раз? Возможно, на первый взгляд этот вопрос имеет и не столь очевидное отношение к поиску, но на самом деле это так и есть. Во-первых, нужно провести поиск по регистрационным файлам и найти строки, в которых есть записи о том, что кто-то вызывал статью. Во-вторых, нужно найти в этих строках название вызванной статьи. В-третьих, нужно отслеживать, насколько часто вызывалась каждая статья. Вот как выглядит пример строки одного из этих файлов, в которой сделаны переносы, чтобы она поместилась на странице этой книги, но на самом деле это одна длинная строка файла:

```
c80-216-32-218.cm-upc.chello.se - - [08/Oct/2006:06:37:48 -0700] "GET  
/ongoing/When/200x/2006/10/08/Grief-Lessons HTTP/1.1" 200 5945 http://www.tbray.org/ongoing/  
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
```

Если прочитать ее слева направо, то можно узнать следующее:

Кто-то в Швеции, из организации по имени chello,
не предоставивший ни своего имени, ни пароля,
зашел на мой веб-блог рано утром 8 октября 2006 года (мой сервер расположен в часовом поясе, имеющем разницу в семь часов с Гринвичем)
и запросил ресурс под названием /ongoing/When/200x/2006/10/08/Grief-Lessons,
используя протокол HTTP 1.1;
запрос был успешным и вернул 5945 байт;
посетитель перешел по ссылке из домашней страницы моего блога
и использовал Internet Explorer 6, запущенный под Windows XP.

Это пример той самой нужной мне строки: одной из тех, в которых записаны реальные вызовы статьи. Существует множество других строк, в которых записаны вызванные таблицы стилей, сценарии, изображения и т. д., а также атаки, предпринятые злоумышленниками. Нужные мне виды строк можно опознать по тому факту, что имя статей начинается с фрагмента /ongoing/ When/ и продолжается элементами десятилетия, года, месяца и дня.

Поэтому в первую очередь нужно найти строки, в которых содержится что-либо подобное следующему:

```
/ongoing/When/200x/2006/10/08/
```

На каком бы языке вы ни программировали, на создание кода посимвольного соответствия этому шаблону можно потратить уйму времени, поэтому лучше воспользоваться регулярными выражениями.

Регулярные выражения

Регулярные выражения относятся к особому языку, разработанному специально для поиска в тексте соответствий шаблонам. Если как следует в них разобраться,

можно сэкономить уйму времени и нервов. Я никогда не встречал по-настоящему сильных программистов, не освоивших работу с регулярными выражениями (которые для краткости часто называют `regexp`s). Первая глава, написанная Брайаном Керниганом (Brian Kernighan), посвящена красоте регулярных выражений.

Поскольку имена файлов на моем веб-сайте соответствуют весьма строгому, базирующемуся на дате шаблону, для поиска интересующих меня регистрационных строк можно подобрать очень простое регулярное выражение. Для поиска в регистрационных файлах других веб-сайтов может понадобиться какое-нибудь другое, более сложное выражение. А мы воспользуемся следующим:

`"GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d[\^ .]+ "`

При взгляде на эту строку сразу же становится ясной одна из проблем регулярных выражений: они не относятся к самому удобочитаемому тексту в мире. Их появление в книге «Иdealный код» у ряда читателей может вызвать некоторое недоумение. Давайте на минутку оставим в покое этот вопрос и займемся нашим регулярным выражением. Нам нужно разобраться в смысле лишь этого конкретного регулярного выражения:

`\d`

означает «соответствует любой цифре от 0 до 9»

`[^ .]`

означает «соответствует любому символу, кроме пробела и точки»¹

`+`

означает «соответствует одному или нескольким появлениюм того, что следовало до знака +».

Поэтому сочетание `[^ .] +` означает, что за последней косой чертой должен следовать набор символов, не содержащий точек и пробелов. Наличие пробела *после* знака + означает, что поиск соответствия регулярному выражению должен закончиться, как только будет найден пробел.

Это регулярное выражение не будет соответствовать строке, где имя файла содержит точку. Поэтому оно будет соответствовать `Grief-Lessons`, то есть тому примеру, который я показывал ранее в регистрационном файле, но не соответствовать `IMG0038.jpg`.

Подключение регулярных выражений к работе

Сами по себе регулярные выражения могут, как показано выше, быть использованы для поиска файлов из командной строки. Но оказывается, что большинство современных компьютерных языков позволяют использовать их непосредственно в программном коде. Давайте так и сделаем и напишем программу для

¹ Знатоки регулярных выражений помнят, что точка является символом-заполнителем «любого символа», но куда труднее запомнить, что точка, заключенная в квадратные скобки, утрачивает свое специальное назначение и указывает всего лишь на точку.

распечатки строк, соответствующих выражению, которое должно подсказать программе, записывающей время чьего-то вызова статьи из веб-блога.

Данный пример (как и многие другие примеры этой главы) написан на языке программирования Ruby, поскольку я верю в то, что он, пусть и далек от совершенства, но относится к наиболее легко читаемым языкам.

Если вы не знаете Ruby, его изучение, возможно, повысит вашу программистскую квалификацию. В главе 29 создатель Ruby Юкихиро Мацумото (Yukihiro Matsumoto) (общеизвестный как «Matz») обсуждает некоторые предпочтения, относящиеся к разработке программ, сумевшие привлечь мое внимание, как и внимание многих других программистов к этому языку.

В примере 4.1 показана наша первая Ruby-программа, к которой слева добавлены номера строк. (Все примеры этой главы доступны на веб-сайте издательства O'Reilly, на страницах, посвященных данной книге.)

Пример 4.1. Распечатка строк, относящихся к вызову статьи

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/\d\d\d\d/\d/\d/\d/[^ .]+ }
3     puts line
4   end
5 end
```

Запуск этой программы приведет к распечатке группы строк регистрационного журнала, похожих на первый пример. Разберем эту программу построчно.

Строка 1

Нам нужно считать все входящие строки, не заботясь о том, откуда именно они взялись: из файла, указанного в командной строке, или были переданы по конвейеру из другой программы или с устройства стандартного ввода. Разработчики Ruby твердо верят в то, что программисты, чтобы справиться со стандартной ситуацией, не должны создавать какой-нибудь уродливый код, а наша ситуация как раз и относится к разряду стандартных. Поэтому ARGF — это специальная переменная, представляющая все входящие источники. Если командная строка включает в себя аргументы, ARGF предполагает, что это имена файлов, и программа открывает эти файлы один за другим; если аргументы отсутствуют, программа использует стандартный ввод.

each_line — это метод, который можно вызвать практически с каждым файлоподобным объектом, таким как ARGF. Он считывает строки на входе и по одной передает их «блоку» следующего кода.

Следующий конструктивный элемент языка — do объявляет, что блок, воспринимающий входные данные, занимает место от него и до соответствующего элемента end, а конструкция |line| требует, чтобы метод each_line загружал каждую строку в переменную line перед тем, как передать ее блоку.

Подобная разновидность цикла может удивить тех, кто впервые столкнулся с Ruby, но он отличается краткостью и мощностью и после кратковременной практики становится вполне понятен.

Строка 2

Это самый обычновенный оператор `if`. Единственным новшеством здесь является сочетание `=~`, которое означает «соответствует» и предполагает наличие последующего регулярного выражения. Можно сообщить Ruby, о наличии регулярного выражения, если поместить в его начале и конце по косой черте (слэшу), например `/это-регулярное-выражение/`. Но конкретно наше регулярное выражение изобилует знаками косой черты. Поэтому для использования слэш-синтаксиса их придется «дезактивировать», превратив каждый слэш (`/`) в комбинацию `\/` и изуродовав весь внешний вид. Поэтому в нашем случае используется сочетание `\g`, позволяющее облагородить код.

Строка 3

Теперь мы находимся внутри блока `if`. Итак, если текущая строка, `line`, соответствует регулярному выражению, программа выполняет строку `puts line`, которая распечатывает строку и символ перевода строки.

Строки 4 и 5

Они относятся ко всему предыдущему. Первый структурный элемент `end` завершает `if`, а второй такой же элемент завершает `do`. Строки выглядят несколько несуразно повисшими на дне всего кода, и разработчики языка Python изобрели способ их опустить, в результате чего образцы кода на языке Python смотрятся красивее, чем соответствующий код на языке Ruby.

Итак, мы показали, как регулярные выражения могут использоваться для поиска интересующих нас строк в регистрационном файле. Но на самом деле наш интерес заключается в подсчете вызовов каждой статьи. Сначала нужно идентифицировать названия статей. Пример 4.2 является несколько измененным вариантом предыдущей программы.

Пример 4.2. Распечатка названий статей

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^\s]+) }
3     puts $1
4   end
5 end
```

Изменения незначительны. Во второй строке я добавил пару скобок, выделенных полужирным шрифтом, поставленных вокруг интересующей нас в регулярном выражении части с названием статьи. В третьей строке вместо распечатки всего значения `line` распечатывается только `$1`, что в Ruby (и некоторых других языках, допускающих использование регулярных выражений) означает «первое место регулярного выражения, взятое в скобки». Можно пометить множество других отрезков регулярного выражения и использовать, соответственно, `$2`, `$3` и т. д.

Первые несколько строк, полученные на выходе после запуска этой программы для обработки некоторого объема данных регистрационного файла, выглядят следующим образом:

```
2003/10/10/FooCampMacs  
2006/11/13/Rough-Mix  
2003/05/22/StudentLookup  
2003/11/13/F1yToYokohama  
2003/07/31/PerlAngst  
2003/05/21/RDFNet  
2003/02/23/Democracy  
2005/12/30/Spoisky-Recursion  
2004/05/08/Torture  
2004/04/27/RSSticker
```

Перед тем как перейти к работе по определению популярности различных статей, я хотел бы доказать, что некоторые важные особенности составляют красоту этого кода. Задумайтесь на минуту о том коде, который пришлось бы написать для поиска произвольной части текста, чтобы провести такой же объем работы по сопоставлению и выбору, который делается с помощью взятого в скобки регулярного выражения. Этот объем занимал бы немало строк, в которых было бы и ошибиться. Кроме того, при изменении формата регистрационного файла работа по исправлению кода поиска соответствий шаблону не обходилась бы без ошибок и раздражений.

Если разобраться, то способ работы регулярных выражений также относится к наиболее удивительным вещам компьютерной науки. Оказывается, регулярные выражения могут быть легко превращены в конечные автоматы. Эти автоматы обладают математическим изяществом, а для проверки соответствия тексту, в котором ведется поиск, существуют удивительно эффективные алгоритмы. Самое замечательное заключается в том, что при запуске автомата в тексте, проверяемом на соответствие, требуется лишь один просмотр каждого символа. Суть эффекта в том, что хорошо построенный движок регулярного выражения способен производить сравнение с шаблоном и выборку быстрее практически любого специально созданного кода, даже если тот прошел ручную оптимизацию на языке ассемблера. Вот в этом и заключается красота.

Я считаю, что код, написанный на языке Ruby, тоже довольно привлекателен. Почти каждый символ программы производит полезную работу. Заметьте, что нет ни точек с запятой в конце строк, ни скобок вокруг блока условий, и вместо `puts(line)` можно написать `puts line`. К тому же отсутствует объявление переменных — они только используются. Такие упрощения конструкции языка способствуют появлению более коротких и легких в написании программ, наряду с тем что их (что не менее важно) становится легче читать и понимать.

Если рассуждать в понятиях времени, то регулярные выражения являются выигрышными со всех сторон. По сравнению с эквивалентным по действию кодом, программисту требуется меньше времени для их написания, меньше времени занимает доставка программы тем людям, которые ее ждут, достигается реальная эффективность в использовании компьютерных мощностей, и пользователи программ меньше времени проводят в скучном ожидании.

Ассоциативное устройство хранения

Теперь мы подошли к самой сути проблемы — к вычислению популярности статей. Нам необходимо извлечь названия статей из каждой строки, распознать их, чтобы установить, насколько часто они были востребованы, увеличить количество запросов на единицу и опять сохранить имеющиеся показания.

Это, может быть, и есть основа поисковых моделей: мы начинаем с *ключа* (используемого для поиска — в данном случае речь идет о названии статьи) и ищем значение (которое хотим найти — в данном случае, сколько раз статья была востребована). Вот несколько других примеров.

Ключ	Значение
Слово	Список веб-страниц, содержащих это слово
Учетный номер работника	Личная карточка работника
Номер паспорта	«Истина» или «ложь», указывающая, должен ли владелец паспорта быть подвергнут дополнительному осмотру

Что реально в данной ситуации нужно программисту — для компьютерной науки это довольно давний вопрос: *ассоциативная память*, также известная как *ассоциативное устройство хранения* и другие различные сочетания подобных слов. Замысел состоит в том, чтобы вставить ключ и получить значение. Существуют определенные аппаратные средства, приспособленные именно для этого; они обитают главным образом в глубинах микропроцессоров, обеспечивая быстрый доступ к страничным таблицам и кэш-памяти.

Для нас, программистов, отрадно то, что используя любой современный компьютерный язык, мы получаем доступ к превосходной программной реализации ассоциативной памяти. В разных языках это реализовано под разными названиями. Зачастую — в виде хэш-таблиц; в Java, Perl и Ruby, использующих данную технологию, они называются *хэшами*, *хэш-картами*, или чем-нибудь подобным. В Python они называются *словарями* (*dictionaries*), а в алгебраическом компьютерном языке Maple — просто *таблицами*.

Теперь, если вам не терпится перейти к алгоритмам поиска и написать собственную сверхэффективную поисковую программу, то сказанное далее скорее прозвучит как плохая, нежели хорошая новость. Но стоит все же призадуматься о факторе времени; если воспользоваться встроенным ассоциативным хранилищем данных, объем времени, затрачиваемого на программирование и овладение инструментарием при написании поисковых алгоритмов, стремится почти что к нулю.

По сравнению со встроенной версией, при создании собственной поисковой системы может быть, вам удастся сэкономить толику компьютерного (а стало быть, и пользовательского) времени, но с другой стороны, может быть, и не удастся; не всем людям, пишущим подобные вещи, удается проявить достаточную сообразительность. Эндрю Качлинг (Andrew Kuchling), написавший главу 18 этой книги, был одним из тех, кто предпринял подобную попытку.

Ассоциативные хранилища, играющие весьма важную роль в таких языках с динамическим определением типов, как Ruby и Python, имеют не только встроенную поддержку, но и специальный синтаксис для их определения и использования. Для подсчета уровня популярности статей в примере 4.3 мы воспользуемся имеющимися в Ruby хеш-структурами.

Пример 4.3. Подсчет востребованности статей

```
1 counts = {}
2 counts.default = 0
3
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d[^ .]+) }
6     counts[$1] += 1
7   end
8 end
```

Эта программа не слишком отличается от версии, предложенной в примере 4.2. В первой строке создается пустой хеш под названием `counts`. Во второй строке массиву дается нулевое «значение по умолчанию» (`default value`); объяснения последуют несколько позже.

Затем, в шестой строке, вместо распечатки названия статьи, это название служит в качестве ключа для поиска количества вызовов этой статьи, содержащегося на этот момент в `counts`, добавляет к нему единицу и сохраняет новое значение.

Теперь представим, что случится, если программа видит какое-то название статьи, сохраненное сначала в переменной `$1`. Я мог бы написать код в несколько строк «если существует элемент `counts[$1]`, то к нему следует добавить единицу; в противном случае, значение `counts[$1]` нужно установить в единицу». Создатели Ruby просто ненавидят подобную неуклюжесть; именно поэтому они предоставили для хеша форму записи «значения по умолчанию». Если вы ищете ключ, о котором в хеш ничего не известно, он как бы отвечает: «Есть такой, его значение равно нулю», позволяя написать следующую строку `counts[$1] += 1` и обеспечивая ее безусловную работоспособность.

Изначально проблема была озвучена следующим образом: «Какие из моих статей читались чаще всего?» В этом есть некая неопределенность; давайте интерпретировать это как задачу «распечатать десятку наиболее популярных статей». В результате получится программа, показанная в примере 4.4.

Пример 4.4. Отчет о наиболее популярных статьях

```
1 counts = {}
2 counts.default = 0
3
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d[^ .]+) }
6     counts[$1] += 1
7   end
8 end
9
10 keys_by_count = counts.keys.sort { |a, b| counts[b] <=> counts[a] }
11 keys_by_count[0 .. 9].each do |key|
```

```
12 puts "#{counts[key]}: #{key}"
13 end
```

Хотя, на мой взгляд, десятая строка выглядит менее привлекательно, чем большинство образцов Ruby-кода, но в ней довольно легко разобраться. Метод `keys`, принадлежащий объекту `counts`, возвращает массив, содержащий все ключи хэша. Из-за особенностей реализации хэша ключи хранятся без заранее определенного порядка и возвращаются методом `keys` также в случайном порядке. Поэтому их необходимо отсортировать и сохранить в новом массиве.

В Ruby метод `sort` сопровождается блоком кода, который в данном случае помещен в фигурные скобки. (В Ruby блок можно обозначить либо с помощью служебных слов `do` и `end`, либо скобками `{` и `}`.) Сортировка работает за счет перемещения вперед и назад по сортируемому массиву, передавая пары элементов блоку, который должен возвращать отрицательное число, нуль или положительное число, в зависимости от того, меньше первый элемент второго, равен ему или больше его.

В данном случае мы хотим получить данные из хэша в порядке, определяемом значениями (само по себе счетчиками), а не именами статей (ключами), поэтому нам нужно отсортировать ключи по их значениям. Присмотревшись к коду, можно понять, как он работает. Поскольку наша задача является довольно распространенной, я удивлен, что хэш Ruby не поставляется вместе с методом сортировки по значению — `sort_by_value`.

Поскольку для сортировки используется убывающий порядок, становится безразличным, сколько статей будет найдено, мы знаем, что первые десять записей в хэше `keys_by_count` представляют десятку самых популярных статей.

Теперь, имея массив ключей (в виде названий статей), отсортированный в убывающем порядке по количеству их вызовов, мы можем выполнить поставленную задачу, распечатав первые десять записей. Стока 11 не отличается сложностью, но в ней настал черед замолвить слово о методе `each`. В Ruby вы почти не увидите оператора `for`, поскольку любой объект, чьи элементы захочется перебрать в цикле, оснащен методом `each`, который делает это за вас.

Возможно, при чтении строки 12 у тех, кто не работал с Ruby, возникнут некоторые затруднения из-за применения синтаксиса `#{}`, но в нем вполне можно разобраться.

Итак, в решении первой задачи можно констатировать победу. Для вполне понятного кода понадобилось всего 13 строк. Но опытные Ruby-программисты скажут, что последние три строки в одну.

Запустим эту программу и посмотрим на результаты ее работы. Вместо того чтобы запускать ее для обработки всех 28 Гб, применим ее к более скромному объему данных: почти что к 1,2 миллиона записей, занявших 245 Мб.

```
~/dev/bc/ 548> zcat ~/ongoing/logs/2006-12-17.log.gz | \
    time ruby code/report-counts.rb
4765: 2006/12/11/Mac-Crash
3138: 2006/01/31/Data-Protection
1865: 2006/12/10/EMail
1650: 2006/03/30/Teacup
1645: 2006/12/11/Java
1100: 2006/07/28/Open-Data
```

```
900: 2006/11/27/Choose-Relax  
705: 2003/09/18/NXML  
692: 2006/07/03/July-1-Fireworks  
673: 2006/12/13/Blog-PR  
    13.54real    7.49 user    0.73 sys
```

Этот запуск состоялся на моем 1,67 ГГц Apple PowerBook. Результаты не вызывают удивлений, но программа кажется несколько медлительной. Нужно ли беспокоиться о ее производительности?

А нужна ли оптимизация?

Я был удивлен, что запуск моего примера показал какую-то непонятную медлительность, поэтому привлек для сравнения очень похожую программу на языке Perl, который, конечно, менее красив, чем Ruby, но работает *очень* быстро. Я был вполне уверен, что работа версии на Perl будет работать вдвое быстрее. Так стоит ли пытаться оптимизировать программу?

И опять нам нужно подумать о времени. Конечно, не исключается возможность заставить все работать быстрее и сократить таким образом время выполнения программы и время, проводимое пользователем в томительном ожидании, но на это нужно потратить рабочее время программиста, а следовательно, и время пользователя на ожидание, пока программист напишет эту программу. Для большинства случаев, как подсказывает моя интуиция, 13,54 секунды, потраченные на обработку недельных данных, — вполне приемлемый показатель, поэтому я констатирую победу. Но, предположим, мы начали получать неудовлетворительные отзывы от работающих с программой людей и захотели бы ускорить ее работу.

Просматривая пример 4.4, можно заметить, что программа разделена на две отдельные части. Сначала она читает все строки и сводит вызовы в таблицу; затем она сортирует их, чтобы найти десятку самых частых вызовов.

Здесь просматривается явная возможность для оптимизации: к чему заниматься сортировкой всех записей о вызовах, когда нужны только десять самых частых вызовов? Нам вполне по силам написать небольшой фрагмент кода для однократного прохода массива и выбора десяти элементов с наивысшими значениями.

Поможет ли это? Я узнал это, вооружившись программой для определения времени, затрачиваемого на выполнение этих двух задач. Ответ (усредненный по нескольким запускам) — 7,36 секунды по первой части и 0,07 секунды по второй, должен сказать: «Нет, не поможет».

Может быть, стоит попытаться оптимизировать первую часть? Наверное, нет. Ведь все, что в ней делается, — сопоставление регулярному выражению, хранение и извлечение данных с использованием хэша, относится к наиболее труднооптимизируемым частям Ruby.

Итак, затея с заменой сортировки, скорее всего, приведет к тому, что программист потратит время впустую на разработку, а заказчик кода — на ожидание, без какой-либо значимой экономии компьютерного и пользовательского времени. К тому же опыт подскажет, что вы не сможете опередить Perl в решении

этого типа задач, поэтому ускорение, которое вы собираетесь получить, будет весьма незначительным.

Мы только что завершили создание программы, которая делает нечто полезное, и оказалось, что с поиском мы тоже уже все сделали. Но мы так, собственно, и не коснулись написания самих поисковых алгоритмов. Давайте этим и займемся.

НЕМНОГО ИСТОРИИ О ПОИСКЕ СООТВЕТСТВИЯ

В духе уважительного отношения надо отдать должное тому, что реальные достижения в сканировании видимых текстовых строк с использованием регулярных выражений и контекстно-адресуемого хранилища в целях получения результатов впервые были популяризованы в языке программирования awk, название которого составлено из первых букв фамилий его изобретателей Ахо (Aho), Вайнбергера (Weinberger) и Кернигана (Kernighan).

Разумеется, эта работа была основана на радикальной на то время философии Unix, обвязанной своим появлением в основном Ричи (Ritchie) и Томпсону (Thompson), что данные должны храниться в основном в файлах, в текстовых строках, и в какой-то мере подтверждала эту философию.

Ларри Уолл (Larry Wall) воспользовался идеями, лежащими в основе awk, и, будучи автором языка Perl, превратил их в высокопроизводительное, эффективное при разработке программ, универсальное инструментальное средство, не удостоившееся заслуженной чести. Оно послужило тем самым элементом, который связал воедино мировые Unix-системы, а впоследствии и большую часть сетей Интернет первого поколения.

Проблема: Кто выбирал, что и когда?

Запуск пары быстродействующих сценариев, обрабатывающих данные регистрационного файла, показывает, что с 2 345 571 различных хостов пришло 12 600 064 выборок статьи. Предположим, что нас заинтересовало, кто что выбрал и когда. Нас могут интересовать аудиторы, полицейские или рыночные профессионалы.

Итак, проблема состоит в том, чтобы по заданному имени хоста определить, какие статьи с него выбирались и когда. Результат будет в виде списка; если список пуст, значит, статьи не вызывались. Мы уже видели, как встроенный в язык хэш или эквивалентные ему структуры данных дают программисту быстрый и легкий способ хранения и поиска пар ключ–значение. Возникает вопрос, почему бы ими не воспользоваться?

Замечательный вопрос, и мы должны дать этой идее шанс на существование. Существуют опасения, что эти структуры не смогут успешно работать, поэтому на всякий случай нужно держать в голове и некий план Б. Если вы когда-либо изучали хэш-таблицы, то можете припомнить, что в целях ускорения их работы у них должен быть небольшой коэффициент загрузки; иными словами большая часть их должна быть пустой. Однако хэш-таблица, в которой содержится 2,35 миллиона записей, которые по-прежнему должны быть большей частью пустыми, будет претендовать на использование слишком большого объема памяти.

В целях упрощения я написал программу, которая обработала все регистрационные файлы и извлекла все востребованные статьи в простой файл, в каждой

строке которого есть имя хоста, время транзакции и название статьи. Вот как выглядят несколько первых строк:

```
crawl-66-249-72-77.googlebot.com 1166406026 2003/04/08/Riffs
egspd42470.ask.com 1166406027 2006/05/03/MARS-T-Shirt
84.7.249.205 1166406040 2003/03/27/Scanner
```

(Второе поле, десятизначное число, является стандартным Unix/Linux представлением времени в виде количества секунд, прошедших с начала 1970 года.)

Затем я написал простую программу, читающую этот файл и загружающую огромный хэш. Эта программа показана в примере 4.5.

Пример 4.5. Загрузка большого хэша

```
1 class BigHash
2
3   def initialize(file)
4     @hash = {}
5     lines = 0
6     File.open(file).each_line do |line|
7       s = line.split
8       article = s[2].intern
9       if @hash[s[0]]
10         @hash[s[0]] << [ s[1], article ]
11       else
12         @hash[s[0]] = [ s[1], article ]
13       end
14       lines += 1
15       STDERR.puts "Line: #{lines}" if (lines % 100000) == 0
16     end
17   end
18
19   def find(key)
20     @hash[key]
21   end
22
23 end
```

Программа должна быть понятной сама по себе, но строку 15 стоит прокомментировать. При запуске большой программы, на работу которой должно уйти много времени, очень раздражает, когда она часами работает без видимых признаков. А что если что-то идет не так? Если она работает невероятно медленно и никогда не закончит работу? Поэтому в строке 15 после обработки каждого 100 000 строк ввода предусмотрена распечатка успокаивающего отчета о ходе процесса.

Интересно было запустить программу. Загрузка хэша заняла примерно 55 минут процессорного времени, и память, занимаемая программой, разрослась до 1,56 Гб. Небольшой подсчет подсказал, что на запись информации о каждом хосте было затрачено примерно 680 байт, или, при иной нарезке данных, около 126 байт на вызов. Это выглядит устрашающе, но, наверное, вполне приемлемо для хэш-таблицы.

Производительность поиска была великолепной. Я запустил 2000 запросов, половина из которых заключалась в выборке из журнала произвольных, следующих

друг за другом хостов, а другая половина касалась хостов с теми же именами, только в обратном, непоследовательном порядке. Две тысячи запросов были выполнены в среднем за 0,02 секунды, значит, обработка хэша в Ruby позволяет просматривать хэш, содержащий примерно 12 миллионов записей, со скоростью несколько тысяч экземпляров в секунду.

Эти 55 минут на загрузку данных вызывают беспокойство, но для решения этого вопроса существует ряд трюков. Вы можете, к примеру, произвести загрузку всего один раз, затем осуществить последовательный вывод хэша и заново его считать. И я особо не пытался оптимизировать программу. Эта программа была написана легко и быстро, и она с самого начала обладала неплохим быстродействием, значит, ее производительность удовлетворяла понятиям и времени ожидания программы и времени ожидания программиста. Тем не менее я не удовлетворен. У меня возникло такое чувство, что должен быть какой-то способ получения такой же производительности при меньших затратах памяти, меньшем времени запуска или и того и другого. Хотя это потребует написания собственного кода поиска.

Двоичный поиск

Квалификацию специалиста по информатике невозможно получить, не изучив широкий спектр алгоритмов поиска: с использованием простых деревьев, частично упорядоченных бинарных деревьев, хэшей, списков и т. д. Среди этого многообразия мне больше всего нравится двоичный поиск. Давайте попробуем с его помощью решить проблему «кто что вызвал и когда», а затем посмотрим, что придает ему красоту.

Первая попытка воспользоваться двоичным поиском была весьма неутешительной; хотя на загрузку данных затрачивалось на 10 минут меньше, потребности в памяти по сравнению с использованием хэша возросли почти на 100 Мб. Выявились некоторые удивительные особенности реализации массивов в Ruby. Поиск также шел в несколько раз медленнее (но не выходил за рамки нескольких тысяч в секунду), но удивляться тут было нечему, поскольку алгоритм работал в коде Ruby, а не на основе жестко заданной хэш-реализации.

Проблема в том, что в Ruby все представлено в виде объектов, и массивы — это довольно-таки абстрактные вещи, обладающие массой встроенной магии. Поэтому давайте реализуем программу на Java, где целые числа таковыми и являются, а массивы обладают незначительным количеством дополнительных свойств¹.

Ничто не может быть концептуально проще двоичного поиска. Вы делите пространство поиска на две половины и смотрите, где вести поиск, в верхней или в нижней половине; затем упражнение повторяется до тех пор, пока все не будет сделано. Интересно, что существует множество неверных способов

¹ Рассмотрение двоичного поиска во многом позаимствовано из моей статьи 2003 года «On the Goodness of Binary Search», доступной на веб-сайте <http://www.tbray.org/ongoing/When/200x/2003/03/22/Binary>.

воплощения этого алгоритма в программном коде, а ряд широко публиковавшихся версий содержит ошибки. Реализация, упомянутая в статье «On the Goodness of Binary Search» и показанная на Java в примере 4.6, основана на одном из вариантов, которому я научился у Гастона Гоннета (Gaston Gonnet), ведущего разработчика предназначенного для символьической логики языка Maple, который сейчас является профессором компьютерных наук швейцарского федерального института технологий (ETH) в Цюрихе.

Пример 4.6. Двоичный поиск

```

1 package binary;
2
3 public class Finder {
4     public static int find(String[] keys, String target) {
5         int high = keys.length;
6         int low = -1;
7         while (high - low > 1) {
8             int probe = (low + high) >> 1;
9             if (keys[probe].compareTo(target) > 0)
10                 high = probe;
11             else
12                 low = probe;
13         }
14         if (low == -1 || keys[low].compareTo(target) != 0)
15             return -1;
16         else
17             return low;
18     }
19 }
```

Программа содержит следующие ключевые моменты.

- В строках 5–6 обратите внимание, что верхняя — `high` и нижняя — `low` границы установлены за пределами обоих концов массива, поэтому ни одна из них изначально не является правильным индексом. Тем самым исключаются любые граничные случаи.
- Цикл, который начинается в строке 7 и работает до тех пор, пока верхняя и нижняя границы не сомкнутся; но средства тестирования, позволяющие понять, что цель найдена, отсутствуют. Подумайте минутку, согласны ли вы с таким решением, а мы вернемся к этому вопросу чуть позже.

У цикла имеется два инварианта. `low` либо равен `-1`, либо указывает на что-либо меньшее или равное целевому значению. А значение `high` либо выходит за верхнюю границу массива, либо указывает на что-либо заведомо большее, чем целевое значение.

- Стока 8 представляет особый интерес. В более ранней версии она выглядела следующим образом:

```
probe = (high + low) / 2;
```

Но в июле 2006 года Java-гуру Джош Блох (Josh Bloch) показал, как при некоторых, не вполне понятных обстоятельствах этот код приводит к целочисленному переполнению (см. <http://googleresearch.blogspot.com/2006/06/extraread-all-about-it-nearly.html>). Тот факт, что спустя многие десятилетия

от момента зарождения компьютерной науки мы продолжаем находить ошибки в основополагающих алгоритмах, конечно, отрезвляет. (Этот вопрос рассматривается также Альберто Савойя (Alberto Savoia) в главе 7.)

Здесь поклонники Ruby наверняка отметят, что современные динамичные языки типа Ruby и Python сами отслеживают целочисленное переполнение и поэтому свободны от подобной ошибки.

- В связи с наличием инвариантов цикла, как только он будет завершен, мне нужно проверить значение `low` (строки 14–17). Если оно не равно `-1`, значит, либо оно указывает на что-то, соответствующее целевому значению, либо целевое значение здесь отсутствует.

Загрузка Java-версии занимает лишь шесть с половиной минут, и она успешно работает, используя менее 1 Гб памяти. К тому же, несмотря на имеющиеся в Java по сравнению с Ruby трудности измерения времени работы процессора, какой-либо заметной задержки с запуском тех же 2000 поисковых задач не было.

Сильные и слабые стороны двоичного поиска

У двоичного поиска есть ряд существенных преимуществ. Прежде всего, его производительность составляет $O(\log_2 N)$. Люди зачастую не представляют себе, насколько это мощное средство. На 32-битном компьютере самое большое значение \log_2 с которым когда-либо придется иметь дело, будет равно 32 (аналогично для 64-битного компьютера это значение будет равно 64), и любой алгоритм, который достигает верхней границы за несколько десятков шагов, будет «вполне приемлем» для многих сценариев реальных ситуаций.

Второе преимущество заключается в краткости и простоте кода двоичного поиска. Код, обладающий такими качествами, можно признать красивым по ряду причин. Возможно, наиболее важной является простота понимания, ведь разобраться в коде порой труднее, чем его написать. К тому же в компактном коде меньше мест, куда могут вкрасться ошибки, он лучше вписывается в наборы инструкций, I-кэши и JIT-компиляторы, в силу чего предрасположен к быстрому выполнению.

Третье преимущество состоит в том, что после получения отсортированного массива надобность в дополнительных индексных структурах отпадает, и пространство памяти используется двоичным поиском весьма эффективно.

Большим недостатком двоичного поиска является то, что данные должны содержаться в памяти в определенном порядке. Существуют наборы данных, для которых это условие невыполнимо, но их количество не столь велико, как может показаться. Если вы считаете, что данных слишком много, чтобы они могли поместиться в память, обратите внимание на сегодняшние цены на блоки оперативной памяти и устраните дефицит. Любая стратегия поиска, требующая обращения к диску, ведет к значительным усложнениям и во многих случаях замедляет процесс.

Предположим, требуется обновить набор данных. Можно подумать, что это будет препятствовать двоичному поиску, поскольку вам нужно обновить в памяти огромный непрерывный массив. Оказывается это проще, чем вы думаете. На самом деле память, занимаемая программой, произвольным образом разбросана по всей физической оперативной памяти компьютера, а страничная система доступа, которую программным путем обеспечивает операционная система, придает ей последовательный вид; тот же трюк вы можете проделать с собственными данными.

Кто-то может утверждать, что поскольку для хэш-таблицы время поиска составит $O(1)$, это должно быть лучше, чем время поиска при двоичном поиске, составляющее $O(\log_2 N)$. На самом деле отличие может быть не столь разительным; можете поэкспериментировать и произвести некоторые замеры. Также нужно взять в расчет, что реализовать хэш-таблицы с необходимым кодом разрешения коллизий значительно сложнее. Я не хочу становиться догматиком, но в последние годы стал использовать к проблемам поиска следующие подходы:

1. пытаться решить их, используя встроенные в язык хэш-таблицы;
2. затем пытаться решить их с помощью двоичного поиска;
3. и только потом с большой неохотой приступать к рассмотрению других, более сложных вариантов.

Выход из цикла

Кое-кто смотрел на мой алгоритм двоичного поиска и спрашивал, почему цикл всегда дорабатывает до конца, не осуществляя проверку на нахождение цели. На самом деле именно так и должно быть; математические выкладки не вписываются в тему этой главы, но если приложить немного усилий, можно выработать некое интуитивное чувство подобных ситуаций — это та самая разновидность интуиции, которая наблюдалась мною у многих великих программистов, с которыми мне доводилось работать.

Подумаем о самом ходе цикла. Предположим, в массиве содержится n элементов, где n представляет собой достаточно большое число. Шансы на то, что цель будет найдена при первом же проходе, равны $1/n$, то есть совсем невелики. Следующий проход (после того как поле поиска будет сужено вдвое) даст шансы $1/(n/2)$ — которые по-прежнему невелики — и т. д. Фактически шансы достичь цели становятся более-менее существенными только после того, как число элементов сократится до 10 или 20, что, к слову сказать, может произойти только за последние четыре прохода цикла. А для случаев, когда поиск будет безуспешным (что часто случается во многих приложениях), подобные дополнительные проверки будут и вовсе нерациональными.

Чтобы определить, когда вероятность достижения цели достигнет 50 %, вы можете произвести математические вычисления, но лучше задайтесь вопросом: стоит ли усложнять каждый шаг алгоритма $O(\log_2 N)$, когда шансы на выигрыш могут появиться только в небольшом количестве самых последних проходов?

Следует понять, что правильно реализованный двоичный поиск представляет собой процесс, проходящий в два этапа. Сперва нужно написать эффективный

цикл, который правильно расставляет нижнюю (*low*) и верхнюю (*high*) границы, а затем добавить простую проверку на то, удалось достичь цели или нет.

Ищите в большом

Задумываясь о поиске, многие представляют себе поиск в Интернете, который предоставляется Yahoo!, Google и их конкурентами. Хотя вездесущий поиск в сети — вещь новая, но порядок поиска по всему тексту, на котором он основан, новизной не отличается. Большинство основополагающих статей были написаны Джеральдом Салтоном (Gerald Salton) из Корнуэльского университета в далеких 60-х годах прошлого века. Основная технология индексирования и поиска больших объемов текста с тех пор не претерпела существенных изменений. Изменилась лишь ранжировка полученного результата¹.

Поиск с постингом

Стандартный подход к полнотекстовому поиску основан на понятии *постинга* — небольшой записи фиксированного размера. Для построения индекса считываются все документы, и для каждого слова создается постинг, в котором есть сведения, что слово *x* появляется в документе *y* с позиции *z*. Затем все слова проходят общую сортировку, чтобы для каждого уникального слова имелся список постингов, каждый из которых представлен парой чисел, состоящих из идентификатора документа и смещения в тексте этого документа.

Поскольку постинги имеют небольшой фиксированный размер, а их количество может неограниченно разрастаться, вполне естественно воспользоваться двоичным поиском. Я не знаю, как конкретно устроены Google или Yahoo!, но я не удивлюсь, если услышу, что десятки тысяч их компьютеров тратят массу времени, проводя двоичный поиск в громадных массивах постингов.

Люди, разбирающиеся в поиске, несколько лет назад дружно посмеивались, когда количество документов, которые Google объявлял охваченными поиском, вплотную подошло к двум миллиардам, а за несколько лет это количество неожиданно резко возросло и продолжало расти и дальше. Видимо, специалисты Google перевели идентификаторы документа во всех постингах с 32 на 64-разрядные числа.

Ранжировка результатов

Поиск в списке постингов заданного слова для определения, какие документы его содержат, не такая уж большая премудрость. Если немного поразмыслить,

¹ Обсуждение полнотекстового поиска во многом позаимствовано из серии моих статей 2003 года «On Search», доступной по адресу <http://www.tbray.org/ongoing/When/200x/2003/07/30/OnSearchTOC>. Серия достаточно широко раскрывает тему поиска, включая вопросы пользовательского опыта, контроля качества, обработки текста, написанного обычным языком, искусственного интеллекта, интернационализации и т. д.

то объединение списков в запросы с использованием операторов AND и OR и поиск по фразам тоже не представляются трудной задачей, по крайней мере, умозрительно. Куда труднее отсортировать список результатов, чтобы самые ценные результаты оказались вверху. В компьютерной науке есть подраздел, называемый информационный поиск – Information Retrieval (для краткости – IR), который почти полностью сконцентрирован на решении этой проблемы. И до недавнего времени результаты были весьма скромными.

Поиск в Интернете

Поисковая система Google и ее конкуренты были в состоянии выработать неплохие результаты, несмотря на невообразимо огромные наборы данных и количество пользователей. Когда я говорю «неплохие», то имею в виду, что самые ценные результаты появляются ближе к вершине списка результатов, а сам список появляется достаточно быстро.

Достижение высококачественных результатов складывается из многих факторов, наиболее значительный из которых, называемый в Google ранжировка страниц – *PageRank*, основан главным образом на подсчете ссылок: страницы, на которые указывает множество гиперссылок, считаются самыми популярными и становятся победителями именно по этому показателю.

Фактически от этого создается впечатление хорошей работы. Но за этим следует ряд интересных наблюдений. Во-первых, до появления PageRank лидерами на пространстве поисковых машин были представления по потребностям, такие как в Yahoo! и DMoz, которые работали по принципу деления результатов по категориям; поэтому, похоже, есть основания утверждать, что намного полезнее знать, насколько информация популярна, чем то, о чём в ней идет речь.

Во-вторых, PageRank подходит лишь для тех коллекций документов, которые насыщены прямыми и обратными ссылками друг на друга. На данный момент определены две коллекции таких документов: Всемирная сеть Интернет – World Wide Web и собрание экспертных обзоров научных изданий (для которых методы, подобные PageRank, применяются десятки лет).

Способность больших поисковых машин расширять рабочее поле с ростом объема данных и количества пользователей производила довольно сильное впечатление. Основанием для нее послужило массированное применение параллельного принципа: штурм крупных проблем с помощью огромного количества небольших компьютеров вместо применения нескольких больших компьютерных систем. Постинги тем хороши, что каждый из них независим от остальных, поэтому они сами собой напрашиваются на применение параллельного подхода.

Например, индекс, полученный на основе проведения двоичного поиска постингов, легко поддается разделению. В индексе, содержащем только английские слова, можно легко создать 26 разделов (для этого используется рабочий термин *shards* (осколки)), по одному для всех слов, начинающихся с каждой буквы алфавита, и каждый из них можно копировать сколько угодно. Тогда огромный объем запросов на словарный поиск может быть распределен по неограниченно большому кластеру взаимодействующих поисковых узлов.

Остаются проблемы объединения результатов поиска по нескольким словам или фразам, и тут уже понадобятся по-настоящему новые подходы, но как можно распараллелить основную функцию поиска по словам, понять нетрудно.

В данном обсуждении несколько несправедливо обойдено стороной множество важных вопросов, особенно это относится к противодействию промышляющим в Интернете злоумышленникам, настойчиво пытающимся перехитрить алгоритмы поисковых машин, чтобы получить некие коммерческие выгоды.

Вывод

Довольно трудно представить себе какое-нибудь компьютерное приложение, которое бы не касалось хранения данных и их поиска на основе содержимого. Весьма характерным примером может послужить наиболее популярное в мире компьютерное приложение — поиск в Интернете.

В данной главе рассмотрены некоторые вопросы, в которых намеренно обойдена традиционная область «базы данных» и вся сфера поисковых стратегий, использующих внешнее хранилище данных. Идет ли речь об одной строке текста или о нескольких миллиардах веб-документов, главным все же является сам поиск. С точки зрения программиста следует отметить, что, кроме всего прочего, создание поисковых систем того или иного вида — занятие увлекательное.

5 Правильный, красивый, быстрый (именно в таком порядке): уроки разработки XML-верификаторов

Элиот Расти Гарольд (Elliotte Rusty Harold)

Этот рассказ относится к двум подпрограммам входной проверки XML, в первой из которых используется JDOM, а во второй – XOM. Я принимал непосредственное участие в разработке кодов обеих подпрограмм, и хотя основы этих двух программных кодов ни в чем не пересекаются и общие строки в них не используются, идеи, заложенные в первом из них, несомненно перетекают и во второй. На мой взгляд, код постепенно становится более привлекательным и, безусловно, более быстрым.

Скорость была ведущим фактором каждого последовательного улучшения, но в данном случае увеличение скорости сопровождалось и приданiem коду большей красоты. Я надеюсь развеять миф о том, что быстрый код должен быть трудно читаемым и уродливым. Напротив, я верю в то, что зачастую это совсем не так, и совершенствование красоты *ведет к увеличению быстродействия*, особенно если учитывать влияние современных оптимизирующих и синхронных компиляторов, RISC-архитектуры и многоядерных процессоров.

Роль XML-проверки

Функциональная совместимость XML достигается за счет строгого придерживания определенных правил, предписывающих, что может, а что не может появляться в XML-документе. Наряду с некоторыми несущественными исключениями, построенный по этим правилам обработчик должен справляться с любым правильно оформленным XML-документом и в состоянии распознать неверно оформленные документы (и не предпринимать попыток их обработки). Тем самым гарантируется высокая степень способности взаимодействия между платформами, парсерами и языками программирования. Не нужно беспокоиться

о том, что ваш парсер не прочитает мой документ, поскольку он написан на Си и запущен под Unix, а мой был написан на Java и запущен под Windows.

Полное соблюдение правил XML обычно предусматривает две избыточные проверки данных.

1. Проверку правильности на входе. В процессе чтения XML-документа парсер проверяет его на правильность оформления и дополнительно на достоверность. Проверка *правильности оформления* имеет отношение исключительно к синтаксическим требованиям, таким как: есть ли у начального тега соответствующий ему конечный тег. Это требование распространяется на все XML-парсеры. Достоверность означает наличие только тех элементов и атрибутов, которые конкретно перечислены в определении типа документа (Document Type Definition, DTD) и появляются только в надлежащих местах.
2. Проверку правильности на выходе. При генерации XML-документа с помощью такого XML API, как DOM, JDOM или XOM, парсер проверяет все строки, проходящие через API, чтобы гарантировать их отношение к XML.

Хотя входящая проверка определена в XML-спецификации более основательно, проверка на выходе может быть не менее важной. В частности, это имеет существенное значение для отладки и обеспечения корректности кода.

Проблема

Самый первый бета-выпуск JDOM не проверял строки, используемые для создания имен элементов, текстового содержимого или, по большому счету, чего-нибудь еще. Программы могли свободно создавать имена элементов, содержащие пробелы, комментарии, завершающиеся дефисом, текстовые узлы, содержащие нулевые строки, и другое неправильно оформленное содержимое. Поддержка правильности сгенерированного XML была полностью оставлена разработчику программ на стороне клиента.

Это не давало мне покоя. Хотя XML-формат проще некоторых альтернативных систем, но не настолько, чтобы быть абсолютно понятным без усвоения секретов его спецификации, к примеру, какие кодировки Юникода могут, а какие не могут использоваться в XML-именах и текстовом содержимом.

JDOM предназначалась в качестве API, несущего XML в массы, такого API, который, в отличие от DOM, не требовал двухнедельного обучения и высокоплачиваемого преподавателя, чтобы научиться его правильно использовать. Чтобы это позволить, JDOM должен был, насколько возможно, избавить программиста от необходимости разбираться в XML. При соответствующей реализации JDOM не оставлял программисту возможности совершить ошибку.

Существует множество способов, позволяющих JDOM добиться такого результата. Некоторые из них вытекают непосредственно из используемой модели данных. Например, в JDOM исключается перекрытие элементов (`<p>Sally said, <quote>let's go the park.</p>. Then let's play ball.</quote>`). Поскольку

в основе внутреннего представления JDOM лежит дерево, то генерация такой разметки в ней просто невозможна. Тем не менее для соблюдения ряда следующих ограничений без проверки, видимо, не обойтись.

- Название элемента, атрибута или инструкции по обработке должно быть допустимым XML-именем.
- Локальные имена не должны содержать двоеточия.
- Пространства имен атрибутов не должны конфликтовать с пространством имен их родительских элементов или сестринских атрибутов.
- Каждый суррогатный символ Юникода должен появляться в виде пары, содержащей его первый байт, за которым следует его второй байт.
- Данные инструкции обработки не содержат двухсимвольную строку ?>.

Когда клиент предоставляет строку для использования в одной из этих областей, она должна пройти проверку на соблюдение важных ограничений. Конкретика изменяется, но основные подходы остаются неизменными.

В соответствии с замыслом главы, я собираюсь проверить действие правил применительно к проверке имен элементов XML 1.0.

В соответствии со спецификацией XML 1.0 (часть которой изложена в примере 5.1) правила предоставлены в нотации Бэкуса–Науэра (Backus–Naur Form, BNF). Здесь #xdddd представляет код Юникода в виде шестнадцатеричного значения dddd. [#xdddd-#xeeee] представляет все коды Юникода от #xdddd до #xeeee.

Пример 5.1. BNF-нотация для проверки XML-имен (в сокращенном виде)

```

BaseChar ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar |
           Extender
Name ::= (Letter | '.' | '_') (NameChar)*
Letter ::= BaseChar | Ideographic
Ideographic ::= [#x4E00-#xF9A5] | #x3007 | [#x3021-#x3029]
Digit ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x0F0-#x0F9]
        | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
        | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
        | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
        | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
Extender ::= #x0087 | #x02D0 | #x02D1 | #x0387 | #x0640 | #xE46 | #x0EC6
           | #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]
           | [#x00D8-#x00F6] | [#x0F8-#x0FF] | [#x0100-#x0131]
           | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
           | [#x0180-#x01C3] ...
CombiningChar ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                 | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x058B-#x058D] | #x058F
                 | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652] | #x0670
                 | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4]
                 | [#x06E7-#x06E8] | [#x06EA-#x06ED]...

```

Полный набор правил займет несколько страниц, поскольку рассмотрению подлежат более 90 000 символов Юникода. В данном примере, в частности, опущены правила для BaseChar и CombiningChar.

Чтобы проверить, какая строка является допустимым XML-именем, необходимо последовательно перебрать все символы в строке и проверить их на приемлемость, определенную в правиле `NameChar`.

Версия 1: Простейшая реализация

Мой начальный вклад в JDOM (показанный в примере 5.2) просто перекладывает проверку правил на имеющийся в Java класс `Character`. Если имя неприемлемо, метод `checkXMLName` возвращает сообщение об ошибке, а если оно отвечает требованиям — возвращает `null`. Сама по себе эта конструкция вызывает сомнения; наверное все-таки она должна вызывать исключение, если имя неприемлемо, и возвращать значение `void` во всех остальных случаях. Позже в этой главе вы увидите, как это будет реализовано в будущих версиях.

Пример 5.2. Первая версия проверки символов имени

```
private static String checkXMLName(String name) {
    // Не может быть пустым или равным null
    if ((name == null) || (name.length() == 0) || (name.trim().equals("")))
        return "Имена XML не могут быть пустыми или равными null.";
    }

    // Не может начинаться с цифры
    char first = name.charAt(0);
    if (Character.isDigit(first)) {
        return "Имена XML не могут начинаться с цифры.";
    }
    // Не может начинаться со знака $
    if (first == '$') {
        return "Имена XML не могут начинаться со знака доллара ($).";
    }
    // Не может начинаться с -
    if (first == '-') {
        return "Имена XML не могут начинаться с дефиса (-).";
    }

    // Проверка на правильность заполнения
    for (int i=0, len = name.length(); i<len; i++) {
        char c = name.charAt(i);
        if ((!Character.isLetterOrDigit(c))
            && (c != '-')
            && (c != '$')
            && (c != '_')) {
            return c + " запрещено использовать в качестве XML-имени.";
        }
    }

    // Если мы добрались до этой строки, значит, с именем все в порядке
    return null;
}
```

Это был довольно простой и понятный метод. К сожалению, его нельзя было считать приемлемым. В частности:

- он допускал имена, содержащие двоеточия. Поскольку JDOM пытается поддерживать корректность пространства имен, этот недостаток должен быть устранен;
- имеющиеся в Java методы `Character.isLetterOrDigit` и `Character.isDigit` не подходят для имеющихся в XML определений букв и цифр. Java рассматривает некоторые символы как буквы, а XML — нет, и наоборот;
- правила в Java от версии к версии склонны к изменениям, а правила в XML — нет.

Тем не менее для первой попытки этот пример можно считать вполне подходящим. Он отлавливал довольно большой процент неправильно составленных имен и не отклонял слишком много правильных. Особенно неплохо он справлялся в самых общих случаях, когда все имена составлялись в ASCII-кодах. Несмотря на это, JDOM стремится к более широкому применению. Нужна более совершенная реализация, реально отвечающая правилам XML.

Версия 2: Имитация BNF-нотации ценой O(N) операций

Мой следующий вклад в JDOM выразился в ручном преобразовании правил BNF в серию операторов `if`-`else`. Полученный результат отображен в примере 5.3. Нетрудно заметить, что эта версия несколько сложнее предыдущей.

Пример 5.3. Проверка символов имени на основе BNF

```
private static String checkXMLName(String name) {
    // Не может быть пустым или равным null
    if ((name == null) || (name.length() == 0)
        || (name.trim().equals("")))) {
        return " Имена XML не могут быть пустыми или равными null. ";
    }

    // Не может начинаться с цифры
    char first = name.charAt(0);
    if (!isXMLNameStartCharacter(first)) {
        return " Имена XML не могут начинаться со знака \'"
            + first + "\'";
    }

    // Проверка на правильность заполнения
    for (int i=0, len = name.length(); i<len; i++) {
        char c = name.charAt(i);
        if (!isXMLNameCharacter(c)) {
            return " Имена XML не могут содержать символ \'"
                + c + "\'";
        }
    }

    // Если мы добрались до этой строки, значит, с именем все в порядке
}
```

```

    return null;
}

public static boolean isXMLNameCharacter(char c) {
    return (isXMLLetter(c) || isXMLDigit(c) || c == '.' || c == '-')
        || c == '_' || c == ':' || isXMLCombiningChar(c)
        || isXMLExtender(c));
}

public static boolean isXMLNameStartCharacter(char c) {
    return (isXMLLetter(c) || c == '_' || c == ':');
}

```

Показанный в примере 5.3 метод checkXMLName вместо повторного использования Java-методов Character.isLetterOrDigit и Character.isDigit доверяет проверку методам isXMLNameCharacter и isXMLNameStartCharacter. Эти методы передают полномочия дальше — методам соответствия другим BNF-правилам для различных типов символов: букв, цифр, комбинированным символам и расширителям. В примере 5.4 показан один из таких методов, isXMLDigit. Обратите внимание, что этот метод учитывает не только цифры ASCII-кода, но и другие цифровые символы, включенные в Юникод 2.0. Методы isXMLLetter, isXMLCombiningChar и isXMLExtender следуют тому же шаблону. Просто они несколько длиннее.

Пример 5.4. Проверка цифровых символов, относящихся к XML

```

public static boolean isXMLDigit(char c) {
    if (c >= 0x0030 && c <= 0x0039) return true;
    if (c >= 0x0660 && c <= 0x0669) return true;
    if (c >= 0x06F0 && c <= 0x06F9) return true;
    if (c >= 0x0966 && c <= 0x096F) return true;

    if (c >= 0x09E6 && c <= 0x09EF) return true;
    if (c >= 0x0A66 && c <= 0x0A6F) return true;
    if (c >= 0x0AE6 && c <= 0x0AEF) return true;

    if (c >= 0x0B66 && c <= 0x0B6F) return true;
    if (c >= 0x0B76 && c <= 0x0BEF) return true;
    if (c >= 0x0C66 && c <= 0x0C6F) return true;

    if (c >= 0x0CE6 && c <= 0x0CEF) return true;
    if (c >= 0x0D66 && c <= 0x0D6F) return true;
    if (c >= 0x0E50 && c <= 0x0E59) return true;

    if (c >= 0x0ED0 && c <= 0x0ED9) return true;
    if (c >= 0x0F20 && c <= 0x0F29) return true;

    return false;
}

```

Этот подход удовлетворял основным целям наращивания возможностей. Он обладал работоспособностью, и его действия были вполне очевидны. Это было очевидное отображение XML-спецификации на программный код. Мы можем

объявить о своей победе и разойтись по домам. Да, насовсем. Именно здесь решил поднять голову уродливый призрак производительности.

Версия 3: Первая оптимизация O(log N)

Как однажды сказал Дональд Кнут (Donald Knuth): «Опрометчивая оптимизация — корень всего зла в программировании». Хотя оптимизация зачастую не настолько нужна, как об этом думают программисты, но она все же играет важную роль, и наш случай как раз из тех немногих, где без нее не обойтись. Анализ показал, что JDOM тратил на осуществление проверки довольно много времени. Каждый содержащийся в имени символ требовал нескольких проверок, и JDOM распознавал недопустимый для имени символ только после его предварительной проверки на принадлежность к каждому разрешенному символу. Следовательно, количество проверок возрастало прямо пропорционально значению указанного кода. Приверженцы проекта допускали недовольные высказывания, что, по большому счету, проверка не столь уж и важна, ее можно выполнять дополнительно или вообще проигнорировать. Теперь я уже не хотел подвергать риску приемлемость имени в угоду быстродействию кода, но, очевидно, решение было за мной, если только кто-нибудь что-нибудь не сделает. К счастью, за дело взялся Джейсон Хантер (Jason Hunter).

Хантер довольно остроумно изменил структуру моего простейшего кода, выдав код, показанный в примере 5.5. Раньше даже в самых распространенных случаях при обработке разрешенного символа требовалось более 100 проверок для каждого из диапазонов недопустимых символов. Хантер заметил, что можно осуществить возврат значения `true` намного быстрее, если одновременно распознавать как разрешенные, так и неразрешенные символы. Особенно выгодно это было в самых распространенных случаях, когда все имена и содержимое представляли собой коды ASCII, поскольку именно эти символы проверялись в первую очередь.

Пример 5.5. Оптимизация проверки цифровых символов

```
public static boolean isXMLDigit(char c) {  
  
    if (c < 0x0030) return false; if (c <= 0x0039) return true;  
    if (c < 0x0660) return false; if (c <= 0x0669) return true;  
    if (c < 0x06F0) return false; if (c <= 0x06F9) return true;  
    if (c < 0x0966) return false; if (c <= 0x096F) return true;  
  
    if (c < 0x09E6) return false; if (c <= 0x09EF) return true;  
    if (c < 0xA66) return false; if (c <= 0xA6F) return true;  
    if (c < 0xAE6) return false; if (c <= 0xAF) return true;  
  
    if (c < 0xB66) return false; if (c <= 0xB6F) return true;  
    if (c < 0xBE7) return false; if (c <= 0xBEF) return true;  
    if (c < 0xC66) return false; if (c <= 0xC6F) return true;  
  
    if (c < 0xCE6) return false; if (c <= 0xCEF) return true;  
    if (c < 0xD66) return false; if (c <= 0xD6F) return true;
```

```
if (c < 0x0E50) return false; if (c <= 0x0E59) return true;  
if (c < 0x0ED0) return false; if (c <= 0x0ED9) return true;  
if (c < 0x0F20) return false; if (c <= 0x0F29) return true;  
  
return false;  
}
```

В предыдущей реализации символ сравнивался со всеми существующими цифрами, и до решения, что символ к цифрам не относится, проверялись даже такие маловероятные символы, как ё и ф. При новом подходе определить, что символ не относится к цифрам, удавалось намного быстрее. Подобные и даже более значительные усовершенствования были реализованы для проверки букв, комбинированных символов и расширителей.

Этот подход не исключал потерю времени на проверку, но привел к ее значительному сокращению, достаточному, чтобы успокоить приверженцев проекта, по крайней мере в отношении имен элементов. Проверка символьных данных, используемых при грамматическом разборе (PCDATA), встроена еще не была, но особой проблемы это не составляло.

Версия 4: Вторая оптимизация: исключение двойной проверки

На данный момент время, затрачиваемое на проверку, было снижено почти в четыре раза и уже перестало вызывать сильное беспокойство. Версия 3, по существу, соответствовала всем требованиям JDOM 1.0. Тем не менее в данной ситуации я решил, что JDOM уже не отвечает критериям качества, и посчитал, что могу ее улучшить. Мои неудачи больше касались вопросов конструкции API, нежели производительности. Беспокойство также вызывала проблема безошибочности, поскольку JDOM по-прежнему проверяла еще не все, что можно, и до сих пор сохранялась возможность (хотя и менее вероятная) использования JDOM для создания искаженных документов. Поэтому я обратился к XOM.

В отличие от JDOM, XOM не допускает компромиссов между проверкой правильности имен и производительностью. По существующему в XOM правилу проверка правильности всегда стоит на первом месте. Тем не менее, чтобы люди отдали предпочтение XOM, а не JDOM, производительность первого должна быть сопоставимой или лучшей, чем у JDOM. Итак, настало время нанести еще один сильный удар по проблеме верификации. Усилия, предпринятые по оптимизации JDOM версии 3, привели к улучшению производительности метода checkXMLName, но при следующей оптимизации я рассчитывал полностью снять эту проблему. На это подталкивали размышления о том, XML-вход не всегда нуждается в проверке, если он исходит из заведомо качественного источника. В частности, пока входная информация попадет к XML-верификатору, XML-парсер уже осуществит многие из необходимых проверок, и в дублировании этой работы нет никакого смысла.

Поскольку конструкторы всегда вели проверку правильности, они снижали быстродействие парсеров, что на практике выливалось в затраты существенной доли времени (зачастую большей его части), требующегося на обработку каждого документа. Эта проблема будет решена путем использования отдельных частей для различных типов входящей информации. Я решил, что конструкторы *не должны* проверять имена элементов, создавая объект из строк, которые в этом документе парсер уже считывал и проверял. И наоборот, конструкторы *должны* проверять имена элементов при создании объекта из строки, переданной библиотечным клиентом. Очевидно, нам понадобятся два различных конструктора: один для парсера, а другой – для всего остального.

Разработчики JDOM рассмотрели возможность такой оптимизации, но были вынуждены ее отложить по причине недостаточной проработанности группы классов (*package*). В JDOM класс SAXBuilder, создающий новый объект Document из SAX-парсера, находится в группе классов org.jdom.input. А Element, Document, Attribute и другие классы узлов – в группе классов org.jdom. Это означает, что все проверочные и непроверочные конструкторы, вызываемые построителем, должны быть общедоступными. Следовательно, другие клиенты могут также осуществить вызов этих конструкторов – речь идет о клиентах, которые не осуществляют соответствующих проверок. Тем самым допускалось производство JDOM дефектного кода XML. Позже, в JDOM 1.0, разработчики развернулись в обратном направлении и решили собрать специальный класс обработки, принимающий все непроверенные входные данные. Этот класс работает быстрее, но открывает потенциальную прореху в верификационной системе. Проблема состояла в искусственном разделении построителя JDOM на входную и основную группу классов.

ПРИМЕЧАНИЕ

Чрезмерное дробление групп классов – это распространенная антимодель, имеющаяся в Java-коде. Зачастую это ставит разработчиков перед не самым лучшим выбором между общедоступностью и ограничением функциональности. Не стоит использовать группы классов только лишь для формирования их структуры. Каждая группа классов должна быть по сути независима от внутренней организации остальных групп.

Если два класса вашей программы или библиотеки должны иметь доступ друг к другу в большей степени, чем к другим, не связанным классам, они должны быть помещены вместе, в одну группу.

В C++ эта проблема искусно решена с помощью дружественных функций. Хотя Java пока не имеет дружественных функций, Java 7 позволяет предоставить расширенный доступ к подгруппам классов, которого нет у обычных элементов программы.

Начиная работу над XOM, я имел возможность учиться на примере JDOM, поэтому поместил входные классы в одну группу с классами основных узлов. Это означало, что я мог обеспечить создание защищенных групповым признаком, не относящихся к проверке методов, доступных парсеру, но не клиентским классам, принадлежащим другим группам.

Используемый в XOM механизм прост и понятен. Каждый класс узла имеет свой частный (*private*), не имеющий аргументов конструктор, наряду с защищенным групповым признаком производственным методом под названием *build*, который вызывает этот конструктор и устанавливает поля без проверки

имен. Все это наряду с имеющим отношение кодом из класса Element демонстрируется в примере 5.6. Фактически XOM относится к пространству имен несколько строже других парсеров, поэтому он обязан осуществлять проверку. Однако он способен опустить множество избыточных проверок.

Пример 5.6. Проверка цифровых символов, основанная на работе парсера

```
private Element( ) {}  
  
static Element build(String name, String uri, String localName) {  
  
    Element result = new Element( );  
    String prefix = "";  
    int colon = name.indexOf(':');  
    if (colon >= 0) {  
        prefix = name.substring(0, colon);  
    }  
    result.prefix = prefix;  
    result.localName = localName;  
    // URI нам нужно проверить именно здесь, поскольку парсеры разрешают  
    // использование относительных URI, которые запрещаются XOM по причинам.  
    // связанным с традиционным XML, если не с чем-нибудь другим. Но нужно  
    // проверить лишь то, что это не абсолютный основной URI. Я не  
    // обязан осуществлять проверку на отсутствие конфликтов.  
    if (!"".equals(uri)) Verifier.checkAbsoluteURIReference(uri);  
    result.URI = uri;  
    return result;  
}
```

Этот подход в известной мере существенно ускорил производительность парсера, поскольку для него не требовался столь же большой объем работы, как для его предшественников.

Версия 5: Третья оптимизация O(1)

После того как я создал конструктор, подробно описанный в предыдущем разделе, и добавил некоторые дополнительные усовершенствования, XOM стала достаточно быстродействующей с учетом всех моих потребностей. Производительность считывания, по сути, ограничивалась только скоростью работы парсера, и в процессе построения документа оставалось совсем немного узких мест.

Однако другие пользователи, чьи потребности отличались от моих, обнаружили ряд различных проблем. В частности, некоторые пользователи создавали собственные построители, которые считывали не принадлежащие XML форматы в дерево XOM. Они не использовали XML-парсер и поэтому не могли воспользоваться ускоренными методами, обходящими верификацию имен. Эти пользователи по-прежнему считали верификацию неким камнем преткновения, хотя и в меньшей, чем раньше, степени.

Я не хотел полностью отключать верификацию, несмотря на просьбы это сделать. Тем не менее назрела явная потребность в ускорении процесса вери-

фикации. Примененный мною способ был из разряда старых классических приемов оптимизации: *табличный поиск*. При табличном поиске вы создаете таблицу, содержащую все ответы для всех известных входных данных. При получении любых входных данных компилятор может просто найти ответ в таблице, не производя никаких вычислений. Для этого понадобится $O(1)$ операция, и скорость ее выполнения близка к теоретическому максимуму. Конечно, весь подвох кроется в деталях.

В Java поиск в таблице проще всего организовать с помощью оператора `switch`. `Javac` компилирует этот оператор в таблицу значений, сохраняемую в байтовом коде. В зависимости от вариантов оператора `switch`, компилятор создает одну из двухбайтовых кодовых инструкций. Если варианты непрерывны (например 72–189 без пропуска каких-нибудь промежуточных значений), компилятор использует более эффективную инструкцию `tableswitch`. Но если пропущено хотя бы одно значение, компилятор использует более замысловатую и менее эффективную инструкцию `lookupswitch`.

ПРИМЕЧАНИЕ

Я не берусь абсолютно гарантировать такое поведение, вполне возможно, что для самых последних виртуальных машин (VM) это утверждение уже несостоит, но оно без всякого сомнения было достоверно для тех VM, которые я проверял в работе.

Для небольших таблиц (не более, чем несколько сотен вариантов) имелась возможность заполнить промежуточные значения значением по умолчанию. Например, простой тест может определить, является ли символ шестнадцатиричной цифрой (пример 5.7). Тест начинается с наименьшего из возможных значений — 0 и заканчивается наибольшим возможным значением — f. Любой символ между 0 и f должен быть включен как вариант.

Пример 5.7. Верификация символов с помощью инструкции `switch`

```
private static boolean isHexDigit(char c) {  
  
    switch(c) {  
        case '0': return true;  
        case '1': return true;  
        case '2': return true;  
        case '3': return true;  
        case '4': return true;  
        case '5': return true;  
        case '6': return true;  
        case '7': return true;  
        case '8': return true;  
        case '9': return true;  
        case ':': return false;  
        case ';': return false;  
        case '<': return false;  
        case '=': return false;  
        case '>': return false;  
        case '?': return false;  
        case '@': return true;  
        case 'A': return true;
```

```
case 'B': return true;
case 'C': return true;
case 'D': return true;
case 'E': return true;
case 'F': return true;
case 'G': return false;
case 'H': return false;
case 'I': return false;
case 'J': return false;
case 'K': return false;
case 'L': return false;
case 'M': return false;
case 'N': return false;
case 'O': return false;
case 'P': return false;
case 'Q': return false;
case 'R': return false;
case 'S': return false;
case 'T': return false;
case 'U': return false;
case 'V': return false;
case 'W': return false;
case 'X': return false;
case 'Y': return false;
case 'Z': return false;
case '[': return false;
case '\\': return false;
case ']': return false;
case '^': return false;
case '_': return false;
case 'a': return true;
case 'b': return true;
case 'c': return true;
case 'd': return true;
case 'e': return true;
case 'f': return true;
}
return false;
}0
```

Все здесь довольно пространно, но плоско. Никакой сложности. Понять, что происходит, совсем не трудно. Однако при всей плоскости операторов `switch` для больших групп вариантов их применение становится проблемным. К примеру, верификация XML-символов предусматривает проверку тысяч вариантов. Я попытался написать оператор `switch`, чтобы справиться с этими более крупными группами, и обнаружил, что в Java на максимальный размер двоичного кода метода существуют ограничения – 64 Кб. Эта ситуация потребовала принятия альтернативного решения.

Хотя компилятор и исполнитель ограничивают размер просматриваемой таблицы, сохраненной в байтовом коде, я мог спрятать ее и в других местах. Сначала я определил простой двоичный формат, по одному байту для каждого из 65536 кодов Юникода в Базовом мультиязыковом наборе кодировок (Basic Multilingual Plane, BMP). Каждый байт содержит восемь однобитных флагов,

которые идентифицируют наиболее важные свойства символа. Например, бит 1 *установлен*, если символ является допустимым в контексте PCDATA, и *сброшен*, если он не является допустимым. Бит 2 *установлен*, если символ может быть использован в качестве XML-имени, и *сброшен*, если не может. Бит 3 *установлен*, если символ может быть первым в XML-имени, и *сброшен*, если не может.

Я написал простую программу для считывания BNF-нотации из XML-спецификации, вычисления значения флага для каждого из 65536 BMP-кодов, а затем сохранения их в одном большом двоичном файле. Этот двоичный файл данных я сохранил вместе с моим исходным кодом и модифицировал задачу для своего компилятора Ant, чтобы скопировать все это в каталог build (пример 5.8).

Пример 5.8. Сохранение и копирование двоичной поисковой таблицы

```
<target name="compile-core" depends="prepare, compile-jaxen">
    <description>Compile the source code</description>
    <javac srcdir="${build.src}" destdir="${build.dest}">
        <classpath refid="compile.class.path"/>
    </javac>
    <copy file="${build.src}/nu/xom/characters.dat"
          tofile="${build.dest}/nu/xom/characters.dat"/>
</target>
```

Оттуда задача jar свяжет поисковую таблицу с компилируемыми файлами .class, поэтому добавления дополнительного файла к установочному пакету или создания дополнительных зависимостей не произойдет. Затем в процессе выполнения программы класс Verifier сможет использовать загрузчик класса для поиска этого файла, как показано в примере 5.9.

Пример 5.9. Загрузка двоичной поисковой таблицы

```
private static byte[] flags = null;

static {
    ClassLoader loader = Verifier.class.getClassLoader();
    if (loader != null) loadFlags(loader);
    // Если это не сработает, следует попробовать другой ClassLoader
    if (flags == null) {
        loader = Thread.currentThread().getContextClassLoader();
        loadFlags(loader);
    }
}

private static void loadFlags(ClassLoader loader) {

    DataInputStream in = null;
    try {
        InputStream raw = loader.getResourceAsStream("nu/xom/characters.dat");
        if (raw == null) {
            throw new RuntimeException("Broken XOM installation: "
                + "could not load nu/xom/characters.dat");
        }
        in = new DataInputStream(raw);
```

```
    flags = new byte[65536];
    in.readFully(flags);
}
catch (IOException ex) {
    throw new RuntimeException("Broken XOM installation: "
        + "could not load nu/xom/characters.dat");
}
finally {
    try {
        if (in != null) in.close();
    }
    catch (IOException ex) {
        // no big deal
    }
}
```

Для этой задачи потребуется 64 Кб динамической памяти. Но для настольного компьютера или сервера это вряд ли составит проблему, к тому же эти данные нужно загрузить лишь однажды. Код старается больше не перезагружать уже загруженные данные.

Когда поисковая таблица загружена в память, проверка любого свойства символа превращается в простую задачу поиска в массиве, за которым следует пара побитовых операций. В примере 5.10 показан новый код проверки имен, не имеющих двоеточия, таких как имена элементов или локальные имена атрибутов. Теперь нужно только найти флаги в таблице и проверить состояние бита, соответствующего требуемому свойству.

Пример 5.10. Проверка имени с использованием поисковой таблицы

```
// константы для битовых флагов в таблице поиска символов
private final static byte XML_CHARACTER = 1;
private final static byte NAME_CHARACTER = 2;
private final static byte NAME_START_CHARACTER = 4;
private final static byte NCNAME_CHARACTER = 8;

static void checkNCName(String name) {

    if (name == null) {
        throwIllegalNameException(name, "NCNames не могут быть null");
    }

    int length = name.length();
    if (length == 0) {
        throwIllegalNameException(name, "NCNames не могут быть пустыми");
    }

    char first = name.charAt(0);
    if ((flags[first] & NAME_START_CHARACTER) == 0) {
        throwIllegalNameException(name, "NCNames не могут начинаться с " +
            "символа " + Integer.toHexString(first));
    }

    for (int i = 1; i < length; i++) {
        char c = name.charAt(i);
        if ((flags[c] & NCNAME_CHARACTER) == 0) {
            throwIllegalNameException(name, "NCNames должны состоять из символов XML_CHARACTER и NAME_CHARACTER");
        }
    }
}
```

```
if ((flags[c] & NCNAME_CHARACTER) == 0) {
    if (c == ':') {
        throwIllegalNameException(name, "NCNames не могут содержать " +
            "двоеточие");
    }
    else {
        throwIllegalNameException(name, "0x"
            + Integer.toHexString(c) + " не используется в именах NCName");
    }
}
}
```

Теперь для проверки символов имен потребуется $O(1)$ операций, а для проверки всего имени — $O(n)$, где n — длина имени. Я думаю, что можно поэкспериментировать с кодом, чтобы улучшить показатели констант, но как это может ускорить работу при осуществлении всех необходимых проверок, понять довольно трудно. Во всяком случае, мы этим еще не занимались.

Версия 6: Четвертая оптимизация: кэширование

Если возможности ускорения работы верификатора уже исчерпаны, то единственная оставшаяся версия уже не сможет или почти не сможет этого сделать. Этот метод был предложен Вольфгангом Хосчеком (Wolfgang Hoschek). Он заметил, что в XML-документе те же самые имена попадаются снова и снова. К примеру, в XHTML-документе всего лишь около ста различающихся имен элементов, из которых несколько десятков используется для большинства элементов (`p`, `table`, `div`, `span`, `strong` и т. д.).

После проверки имени на приемлемость его можно сохранить в каком-нибудь семействе. Когда имя будет замечено еще раз, сначала проверяется его принадлежность к встречавшимся ранее именам, и если проверка будет успешной, оно просто принимается без повторной проверки.

Тем не менее в данном случае следует проявлять особую осторожность. Время поиска некоторых (особенно коротких) имен в таких семействах, как хэш-карта, может быть большим, чем их повторная проверка. Определить это можно только путем сравнения и тщательной оценки совокупности параметров схем кэширования нескольких разных виртуальных машин, используя различные виды документов. Может потребоваться подгонка таких параметров, как размер семейства под разные виды документов, ведь то, что хорошо работает с одним видом документа, может работать хуже с другим. Более того, если кэш совместно используется несколькими потоками, их состязание может стать серьезной проблемой.

Поэтому я до сих пор не применил эту схему для имен элементов. Однако я использовал ее для пространства имен URI (Uniform Resource Identifiers), проверка которых обходится даже дороже, чем проверка имен элементов, а повторяемость бывает и выше. Например, у многих документов есть только одно

пространство имен URI, и только немногие имеют более четырех, поэтому потенциальное преимущество здесь значительно выше. В примере 5.11 показан внутренний класс, который XOM использует для кэширования пространства имен URI после их проверки.

Пример 5.11. Кэш для верификации пространства имен URI

```
private final static class URICache {
    private final static int LOAD = 6;
    private String[] cache = new String[LOAD];
    private int position = 0;

    synchronized boolean contains(String s) {
        for (int i = 0; i < LOAD; i++) {
            // Здесь я предполагаю, что пространство имен изолировано.
            // Обычно так и бывает, но не всегда. Но даже в этом случае
            // отказа в работе не будет. Использование equals( ) вместо
            // == работает быстрее, когда пространство имен URI не
            // изолировано, но медленнее, если оно изолировано.

            if (s == cache[i]) {
                return true;
            }
        }
        return false;
    }

    synchronized void put(String s) {
        cache[position] = s;
        position++;
        if (position == LOAD) position = 0;
    }
}
```

В этом классе есть несколько удивительных особенностей. Во-первых, вместо того чтобы использовать подразумеваемую в данном случае хэш-карту или таблицу, в нем используется массив фиксированного размера с последовательным перебором элементов. Для таких небольших списков постоянные издержки поиска в хэше оказываются медленнее, чем простой перебор элементов массива.

Во-вторых, если массив заполняется, он уже больше не расширяется. Новые данные просто пишутся поверх старых, начиная с первой позиции. Такое поведение может быть подвергнуто резкой критике, поскольку оно способно снизить производительность, хотя работа при этом не нарушится, но значительно замедлится. И все же крайне маловероятно, что при обработке какого-нибудь настоящего XML-документа возникнет такая проблема. Не у многих из них будет более шести пространств имен, и в тех редких случаях, когда такое произойдет, пространства имен обычно локализованы, а не раскиданы по всему документу. Падения производительности, вызванные перезагрузкой массива, будут носить сугубо временный характер.

Я могу представить себе только один случай, при котором статический размер может вызвать реальную проблему – если множества потоков будут одновременно проводить разбор документов, по виду резко отличающихся друг от друга. В таком случае может быть значительно превышено ограничение на шесть пространств имен. В одном из возможных решений вместо этого скорее всего будет использована организация кэша потоковой локальной переменной.

Наши беспокойства значительно возрастают, если придется кэшировать не только пространства имен URI, но и имена элементов. В таком случае количество имен, имеющих меньшую длину, значительно возрастает. Поэтому, наверное, более разумно воспользоваться не простым массивом, а таблицей. Возможно, верификация станет быстрее. Пока я не проводил подобных замеров, необходимых для определения наилучшей конструкции, но планирую сделать это для XOM 1.3.

Мораль всей этой истории

Если у этой истории и есть какая-то мораль, то она заключается в следующем: не сбивайтесь с правильного пути в угоду производительности. Быстродействие кода, проявив немного смекалки, всегда можно повысить. А вот столь же легко исправить неудачную конструкцию вряд ли удастся.

Со временем программы обычно повышают, а не снижают свое быстродействие. Этому, в частности, способствует повышение скорости процессоров, но самое главное не в этом. Куда большая польза получается от улучшенных алгоритмов. Поэтому проектирование программы должно идти по соответствующим канонам. И только потом нужно позаботиться о производительности. Чаще всего с первого же запуска обнаружится, что программа обладает вполне приемлемым быстродействием, и какие-нибудь ухищрения, подобные тем, на которые здесь обращалось внимание, не понадобятся. Тем не менее, если все же придется заняться быстродействием то куда проще сделать быстрее красивый, но не самый быстрый код, чем придать красоту уродливому, но быстродействующему.

6 Платформа для проведения комплексного теста: красота, полученная за счет хрупкости конструкции

Майкл Фезерс (*Michael Feathers*)

Как и у каждого программиста, у меня есть ряд идей, касающихся хороших конструкций. У всех нас эти идеи исходят из практики, возникая в ходе работы. Если появляется соблазн использовать публичную переменную в классе, мы вспоминаем, что публичные переменные чаще всего являются признаком плохой конструкции, а если мы видим наследование реализации, то вспоминаем, что лучше отдать предпочтение не наследованию, а делегированию¹.

Подобные эмпирические правила приносят заметную пользу. Они помогают в процессе работы прокладывать путь сквозь пространство проектных параметров, но, забыв о них, мы вредим самим себе. Забывчивость ведет к тому, что проект, в котором «все сделано правильно», все же не приводит к желаемой цели.

Эти мысли вернули меня в 2002 год, когда Уорд Каннингем (Ward Cunningham) выпустил свое средство тестирования — платформу для проведения комплексного теста (Framework for Integrated Test, FIT). FIT состоит из небольших наборов превосходных Java-классов. Они обходят практический каждый из выработанных Java-сообществом правил разработки, и каждое допущенное в них незначительное отступление является вынужденной мерой. Эти классы резко контрастируют с разработками, которые выполнены строго по правилам.

На мой взгляд, FIT является примером красивого кода. Он наводит на размышление о том, что характер конструкции должен определяться исходя из ситуации.

¹ «Шаблоны проектирования — элементы многократно используемого объектно-ориентированного программного обеспечения» («Design Patterns: Elements of Reusable Object-Oriented Software», Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995.)

В этой главе я предлагаю рассмотреть одну из наиболее ранних из выпущенных версий FIT. Я покажу, как FIT отклонилась от большинства принятых на данный момент в Java и в системе разработки объектно-ориентированных платформ мудрых правил, и расскажу, как FIT спровоцировала меня на пересмотр некоторых из глубоко укоренившихся в моем сознании конструкторских предубеждений. Я не знаю, станете ли вы после прочтения этой главы пересматривать свои собственные взгляды, но тем не менее я приглашаю вас присмотреться к излагаемому материалу. Если повезет, то я сумею выразить все особенности, составляющие конструкцию FIT.

Платформа для приемочных испытаний, выполненная в трех классах

Что такое FIT, можно объяснить достаточно просто. Это небольшая платформа, позволяющая создавать выполняемые тесты приложений в HTML-таблицах. Каждый тип таблицы обрабатывается классом, который называется стендом (*fixture*) и определяется программистом. Когда платформа обрабатывает HTML-страницу, она создает стендовый объект для каждой таблицы, имеющейся на этой странице. Стенд использует таблицу в качестве входных данных для выбранного вами проверочного кода: он считывает значения ячеек, связывается с вашим приложением, проверяет ожидаемые значения и помечает ячейки зеленым или красным цветом, чтобы показать успех или неудачу проверки.

В первой ячейке таблицы указывается имя стендового класса, который будет использован для обработки таблицы. К примеру, на рис. 6.1 показана таблица, которая будет обработана стендом *MarketEvaluation*. На рис. 6.2 показана та же таблица, подвергшаяся обработке FIT; при экранном отображении затененные ячейки будут подсвечены красным цветом, чтобы показать ошибку, возникшую при проверке.

MarketEvaluation				
Symbol	Shares	Price	Portfolio	Attempt Fill?
PLI	1100	1560	no	no

Рис. 6.1. HTML-таблица, отображенная до FIT-обработки

MarketEvaluation				
Symbol	Shares	Price	Portfolio	Attempt Fill?
PLI	1100	1560	no	no expected
				yes actual

Рис. 6.2. HTML-таблица, отображенная после FIT-обработки

Ключевая идея, заложенная в FIT, состоит в том, что документы могут служить в качестве тестов. К примеру, вы можете вставить таблицы в документ с требованиями и пропустить этот документ через FIT, чтобы посмотреть, существует ли характер поведения, определенный в этих таблицах, в вашем программном обеспечении. Этот документ, содержащий таблицы, может быть написан непосредственно в HTML-коде или создан в Microsoft Word или в любом другом приложении, способном сохранять документ в HTML-формате. Поскольку стенд FIT – это часть программного обеспечения, он может вызвать любой фрагмент тестируемого приложения и осуществить такие вызовы на любом уровне. И все это находится под вашим, программистским контролем.

Мне больше не хочется тратить время на объяснения FIT и всего, что с ним связано; всю дополнительную информацию можно получить на веб-сайте, посвященном FIT (<http://fit.c2.com>). Я хочу остановиться на описании конструкции FIT и некоторых воплощенных в ней интересных решений.

Ядром FIT являются всего лишь три класса: Parse, Fixture и TypeAdapter. Их поля и методы и отношения между ними показаны на рис. 6.3.

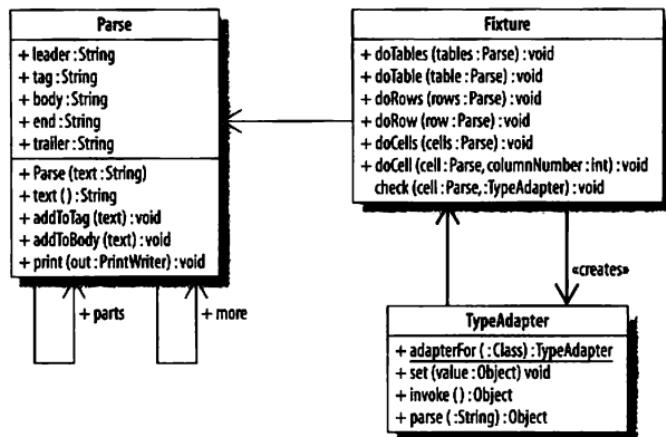


Рис. 6.3. Отношения между FIT-классами

Давайте со всем этим разберемся.

Братьце класс Parse представляет содержащийся в документе HTML. Имеющийся в Parse конструктор принимает строку и рекурсивно выстраивает дерево Parse-объектов, связанных друг с другом полями parts и more. Каждый Parse-объект представляет определенную часть документа: для каждой таблицы, строки и ячейки существует свой собственный Parse.

Класс Fixture обходит дерево Parse-объектов, а TypeAdapter превращает тестируемые значения (числовые, даты/времени и т. д.) в текст и обратно. Стенды (Fixtures) взаимодействуют с тестируемым приложением и помечают отдельные ячейки красным или зеленым цветом, в зависимости от неудачного или удачного прохождения теста. В FIT вся основная часть работы происходит

в подклассах класса `Fixture`. Эти подклассы определяют формат интерпретируемых ими таблиц. Если нужно создать таблицу, состоящую, скажем, из наборов команд, которые должны быть выполнены относительно вашего приложения, вы используете предопределенный класс `ActionFixture`. Если нужно запросить у приложения множество результатов и сравнить их с набором ожидаемых значений, вы используете класс `RowFixture`. FIT предоставляет эти простые подклассы, но наряду с этим она позволяет создавать собственные подклассы `Fixture`.

FIT – весьма востребованная платформа. Я часто ею пользуюсь и не перестаю удивляться тем возможностям, которые предоставляются этими тремя классами. Чтобы проделать сравнимый объем работы, многим платформам требуется задействовать в три или в четыре раза больше классов.

Сложности конструкции платформы

Ну хорошо, все это относилось к архитектуре FIT. А теперь поговорим о ее отличительных особенностях. Разработка платформы не обходится без сложностей. Главные трудности заключаются в отсутствии контроля над тем кодом, который использует вашу платформу. Как только будет опубликован API платформы, вы уже не сможете изменить сигнатуры конкретных методов или изменить их семантику, без необходимости проведения определенной работы с пользователями платформы. К тому же пользователи платформы обычно не желают пересматривать свой код при обновлении платформы, им нужна полноценная обратная совместимость.

Традиционным способом преодоления этой проблемы является соблюдение следующих практических правил.

1. Проявление осторожности при назначении публичной области определения (`public`): следует контролировать область видимости, оставляя общедоступную составляющую платформы незначительной.
2. Использование интерфейсов.
3. Предоставление строго определенных «точек прерывания», которые допускают расширяемость только в специально предназначенных для этого местах.
4. Предотвращение расширения там, где это нежелательно.

Некоторые из этих указаний уже встречались в различных источниках¹, но большая часть из них составляет культуру программирования. Они являются общепринятыми приемами конструирования платформ. Рассмотрим пример, не имеющий отношения к FIT, демонстрирующий эти правила в действии: `JavaMail API`.

¹ «Effective Java», Joshua Bloch, Prentice Hall PTR, 2001. «Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries», Krzysztof Cwalina, Brad Abrams, Addison-Wesley Professional, 2005.

Если требуется получить почту с использованием JavaMail, вам нужно получить ссылку на объект Session, который является классом, имеющим статический метод под названием getDefaultInstance:

```
package javax.mail;
public final class Session
{
    ...
    public static Session getDefaultInstance(Properties props);
    ...
    public Store getStore( ) throws NoSuchProviderException;
    ...
}
```

Класс Session является конечным. В Java это означает, что мы не можем на его основе создать подкласс. Более того, у Session нет публичного конструктора, поэтому мы не можем самостоятельно создать экземпляр класса и вынуждены для этого использовать статический метод.

Как только будет создан объект Session, мы сможем воспользоваться методом getStore и получить объект хранилища — Store, который содержит папки с полученной почтой и ряд других полезных вещей.

Store является абстрактным классом. Платформа JavaMail предоставляет два подкласса на основе Store, но нас, как пользователей, не должно волновать, какой из них нам возвращается; мы просто его принимаем и используем для получения своей почты. Если нам нужно получить уведомление о том, что платформа поменяла подкласс Store, мы можем зарегистрировать принимающего информацию в экземпляре Store, воспользовавшись следующим методом:

```
public abstract class Store extends javax.mail.Service {
    ...
    public void addStoreListener(StoreListener listener);
    ...
}
```

Эта небольшая часть платформы JavaMail является воплощением традиционного стиля построения подобных структур. Класс Session объявляется конечным, чтобы предотвратить создание подклассов. В JavaMail также используется абстрактный класс хранилища — Store, чтобы предоставить место для расширения: здесь может быть создано множество разных хранилищ. Однако Store является охраняемым местом расширения. Способов заставить Session программным путем вернуть какой-нибудь особенный объект Store не существует. Возвращаемый Store может быть сконфигурирован, но только не в программном коде. Платформа ограничила такую возможность.

Тем не менее вы можете зарегистрировать в Store того, кто принимает информацию. Эта регистрация принимающего — небольшая «заштепка», определенная разработчиками платформы, чтобы пользователи могли получать уведомления о происходящем в хранилище, не получая при этом самого подкласса Store.

В итоге JavaMail API очень хорошо справляется со своей защитой. Чтобы им воспользоваться, вам нужно придерживаться четко определенной последовательности действий, а разработчики платформы имеют некую «оболочку»,

внутри которой они могут вносить будущие изменения; они могут вернуться к разработке и внести любые изменения внутри класса Session, не испытывая волнений о каких-нибудь подклассах и о подмене каких-то их частей.

Открытая платформа

FIT является платформой совершенно другого типа. Рассматривая UML-диаграмму на рис. 6.3, вы могли заметить, что практически все, даже данные в ее ядре, является публичным. Если вы просмотрите код других популярных платформ, то не найдете в нем ничего подобного.

Но как это вообще может работать? Ее работа обеспечивается благодаря тому, что FIT является примером открытой платформы. В ней отсутствует ограниченный набор мест расширения. Платформа сконструирована полностью расширяемой.

Рассмотрим класс Fixture. Клиенты класса Fixture используют его напрямую. Они создают экземпляр Fixture, с помощью Parse создают дерево анализа HTML-документа, а затем передают дерево методу doTables. Этот метод, в свою очередь, вызывает для каждой таблицы документа метод doTable, проводя ее через соответствующее поддерево анализа.

Метод doTable выглядит следующим образом:

```
public void doTable(Parse table) {  
    doRows(table.parts.more);  
}
```

А вызываемый из него метод doRows имеет следующий вид:

```
public void doRows(Parse rows) {  
    while (rows != null) {  
        doRow(rows);  
        rows = rows.more;  
    }  
}
```

В свою очередь, метод doRow вызывает метод doCells:

```
public void doRow(Parse row) {  
    doCells(row.parts);  
}
```

И эта последовательность заканчивается методом под названием doCell:

```
public void doCell(Parse cell, int columnNumber) {  
    ignore(cell);  
}
```

Метод ignore просто добавляет к ячейке серый цвет, показывая тем самым, что ячейка была проигнорирована:

```
public static void ignore (Parse cell) {  
    cell.addToTag(" bgcolor=\"#efefef\"");  
    ignores++;  
}
```

По определению не похоже, чтобы класс Fixture вообще занимался чем-то существенным. Все, что он делает, — это обеспечивает прохождение документа

и окрашивает ячейки в серый цвет. Тем не менее подклассы `Fixture` могут подменять любой из этих методов и делать что-нибудь другое. Они могут собирать информацию, сохранять ее, связываться с приложениями во время тестирования и помечать значения ячеек. Класс `Fixture` определяет исходную последовательность прохождения HTML-документа.

Этот образ действий сильно отличается от общепринятого для платформ. Пользователи не «подключаются» к этой платформе, а создают на основе класса свой подкласс и подменяют часть исходных действий. К тому же у этой платформы отсутствует защитная оболочка для ее разработчиков. С технической точки зрения, пользователю нужно лишь вызвать метод `doTables`, но вся последовательность прохождения, от `doTables` до `doCell`, имеет публичную область определения. FIT должна будет всегда придерживаться этой последовательности прохождения. Каким-либо образом ее изменить, не ломая код заказчика, невозможно. С точки зрения традиционной платформы, это плохо, но как это можно оценить при полной уверенности именно в такой последовательности прохождения? В этой последовательности отражаются важные для нас, абсолютно стабильные составные части HTML-кода; довольно трудно представить себе, что код HTML претерпит изменения, способные ее нарушить. И нас такое положение вещей будет устраивать всегда.

Насколько просто может быть устроен HTML-парсер?

В дополнение к тому, что FIT является открытой платформой, в ее конструкции есть еще ряд не менее удивительных вещей. Я уже упоминал, что в FIT весь анализ HTML проводится в классе `Parse`. На мой взгляд, одна из самых симпатичных особенностей класса `Parse` заключается в том, что он выстраивает все дерево вместе со своими конструкторами.

И вот как это работает. Вы создаете экземпляр класса со строкой HTML в качестве аргумента конструктора:

```
String input = read(new File(argv[0]));
Parse parse = new Parse(input);
```

Конструктор `Parse` рекурсивно выстраивает дерево экземпляров `Parse`, где каждый из экземпляров представляет некую часть HTML-документа. Весь анализирующий код размещается внутри конструкторов `Parse`.

Каждый экземпляр `Parse` имеет пять публичных строк и две ссылки на другие `Parse`-объекты:

```
public String leader;
public String tag;
public String body;
public String end;
public String trailer;

public Parse more;
public Parse parts;
```

Когда вы создаете для HTML-документа свой первый экземпляр Parse, то в некотором смысле вы создаете и все остальные экземпляры. С этого момента вы можете переходить по узлам, используя ссылки *тоге* и *parts*.

Анализирующий код класса Parse выглядит следующим образом:

```
static String tags[] = {"table", "tr", "td"};

public Parse (String text) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[]) throws ParseException {
    this (text, tags, 0, 0);
}

public Parse (String text, String tags[], int level, int offset) throws
ParseException
{
    String lc = text.toLowerCase();
    int startTag = lc.indexOf("<" + tags[level]);
    int endTag = lc.indexOf(">", startTag) + 1;
    int startEnd = lc.indexOf("</>" + tags[level]), endTag;
    int endEnd = lc.indexOf(">", startEnd) + 1;
    int startMore = lc.indexOf("<" + tags[level]), endEnd;
    if (startTag<0 || endTag<0 || startEnd<0 || endEnd<0) {
        throw new ParseException ("Can't find tag: " + tags[level].offset);
    }

    leader = text.substring(0,startTag);
    tag = text.substring(startTag, endTag);
    body = text.substring(endTag, startEnd);
    end = text.substring(startEnd,endEnd);
    trailer = text.substring(endEnd);
    if (level+1 < tags.length) {
        parts = new Parse (body, tags, level+1, offset+endTag);
        body = null;
    }
    if (startMore>=0) {
        more = new Parse (trailer, tags, level, offset+endEnd);
        trailer = null;
    }
}
```

Одна из самых интересных особенностей Parse заключается в том, что он представляет весь HTML-документ. Стока *leader* содержит весь текст, предшествующий HTML-элементу, *tag* содержит текст тега, *body* содержит текст между открывающим и закрывающим тегами, а *trailer* содержит весь замыкающий текст. Поскольку в Parse содержится весь текст, вы можете перейти к анализу верхнего уровня и получить его распечатку, воспользовавшись следующим методом:

```
public void print(PrintWriter out) {
    out.print(leader);
    out.print(tag);
    if (parts != null) {
        parts.print(out);
    } else {
```

```

    out.print(body);
}
out.print(end);
if (more != null) {
    more.print(out);
} else {
    out.print(trailer);
}
}
}

```

На мой взгляд, нет более изящных примеров, написанных на Java, чем этот. Долгое время я придерживался правила, что конструкторы не должны продлевать в классе основную часть всей работы. Что их задача ограничивается приведением объекта в должное состояние, а вся реальная работа возлагается на другие методы. Вообще-то осуществление каких-либо значимых операций при создании объекта – явление довольно неожиданное и удивительное. Тем не менее код конструктора класса Parse обладает бесспорным изяществом. В том способе, которым он разбивает HTML-строку на несколько сотен Parse-объектов, а затем восстанавливает ее, используя единственный метод – print, есть по-своему красавая гармония.

Если аналитический и представляющий коды были бы размещены в отдельных классах, платформа могла бы вести обработку различных форматов, таких как XML или RTF; тем не менее решение ограничиться обработкой HTML-формата представляется правильным выбором. При этом платформа получается небольшой по объему и простой в понимании – весь аналитический код занимает всего около 25 строк. Выбор единственного формата придает платформе FIT простоту и надежность. Один из самых ценных уроков конструирования заключается в том, что сохраняя удачные вещи в неизменном виде, можно извлечь немалую пользу.

Еще одна интересная особенность Parse-объектов состоит в том, что для хранения ссылок на своих соседей в них не используются семейства. Они используют parts и more в виде непосредственных ссылок. Это придает коду некоторую схожесть с языком Lisp, но если вы привыкли смотреть на все с функциональной точки зрения, это вполне логично.

В приводимом далее примере подобного стиля программирования принадлежащий классу Parse метод last возвращает последний элемент в последовательности more объекта Parse:

```

public Parse last( ) {
    return more==null ? this : more.last( );
}

```

Опять-таки интерес вызывает тот факт, что все поля объекта Parse являются публичными. Платформа не может каким-то образом воспротивиться желанию пользователя их изменить. Конечно, существуют такие, вполне удобные методы, как addToTag и addToBody, но любой желающий может непосредственно внести изменения в поля, с которыми они работают. FIT предоставляет вам такую возможность, относя издержки на свой собственный счет: будущие версии FIT не могут вот так запросто взять и отменить эту возможность доступа. Это уже не относится к тем вещам, которыми все разработчики платформ могут распоря-

жаться по своему собственному усмотрению, но если вы понимаете, к чему могут привести последствия, значит, выбор был правильным.

Вывод

Многие из нас, работающих в сфере производства программного обеспечения, вынесли свои уроки, пройдя школу ударов судьбы. Мы попадали в такие ситуации, когда ранее созданное нами программное обеспечение не обладало желаемой степенью расширяемости. Со временем наша реакция выразилась в подборке практических методов, направленных на защиту расширяемости путем введения ограничений. Возможно, что при разработке платформы для нескольких тысяч пользователей ничего лучшего вам и не представится, но этот путь не является для вас единственным.

Альтернатива, продемонстрированная FIT, носит радикальный характер: попытаться сделать платформу как можно гибче и компактнее, не разбивая ее на десятки классов, а напротив, проявляя крайнюю осторожность в вопросах внутреннего разбиения каждого класса. Методам придется публичная область определения, чтобы пользователи при необходимости могли отклониться от обычного курса, но наряду с этим принимается ряд жестких решений, таких как выбор HTML в качестве единственной передающей среды для FIT.

При разработке подобной платформы вы сталкиваетесь с совершенно другой областью программного обеспечения. Поскольку классы имеют открытый характер, внести в них изменения в последующих версиях будет совсем непросто, но зато ваше творение может быть достаточно простым и компактным, чтобы склонить людей к созданию чего-нибудь нового.

И это необычное искусство. Мир просто наводнен платформами, которые с трудом поддаются пониманию и выглядят так, как будто в них вложено слишком много сил, настолько много, что весьма трудно обосновать необходимость заново изобретать такое же колесо. Когда же такое происходит, мы отказываемся от массивных платформ, которые теряют важные области применения.

Если отвлечься от самих платформ, то можно многое сказать об адаптации этого «открытого» стиля разработки к небольшим проектам. Если известны все пользователи вашего кода — если, к примеру, все они работают вместе с вами над одним и тем же проектом — вы можете сделать выбор в пользу понижения степени защиты некоторых частей своего кода, чтобы сделать его таким же простым и понятным, как код FIT.

Не весь код имеет отношение к API. Мы легко можем забыть об этом, поскольку у всех нас на слуху ужасные истории (случай, когда одна команда ограничивает возможности другой, внедряя втихаря такие зависимости, от которых невозможно избавиться), но если вы работаете в составе единой команды — если заказчики вашего кода имеют возможность и хотят вносить изменения, когда вы их об этом попросите — вы можете принять другой стиль разработки и стремиться к созданию весьма открытого кода. Решением проблемы может быть защита, основанная на особенностях языка программирования или на

конструкции программного продукта, но сама проблема носит социальный характер, и очень важно рассмотреть, насколько вообще реально ее существование.

Время от времени я показываю код FIT опытному разработчику и получаю от него довольно интересную реакцию. Он сразу же заявляет: «Очень мило, но я никогда не буду создавать код вроде этого». Тогда я прошу его самому себе ответить на вопрос, почему он придерживается такого мнения. На мой взгляд FIT – превосходная платформа, поскольку она способна вызвать подобные вопросы и заставить задуматься над ответами.

7

Красивые тесты

Альберто Савойя (Alberto Savoia)

Большинство программистов набираются опыта, изучая фрагменты кода и оценивая их не только с точки зрения функциональности, но и *красоты*. Обычно код считается красивым, когда он не только оправдывает функциональные ожидания, но и обладает при этом особой элегантностью и экономичностью.

А что можно сказать о тестах для красивого кода — особенно о тех тестах, которые создаются разработчиками или *должны* ими создаваться в процессе отработки программного кода? В данной главе я собираюсь остановиться на тестах, поскольку они могут быть красивы сами по себе. Но важнее все-таки то, что они могут сыграть ключевую роль, помогая создать более красивый код.

Мы поймем, что красота тестов складывается из сочетания многих вещей. В отличие от кода, я не могу себя заставить посчитать красивым любой *отдельный* тест — по крайней мере я не могу рассматривать его так же, как, скажем, процедуру сортировки, и называть красивым. Причина в том, что тестиирование по своей природе — это решение комплексных и исследовательских задач. Каждый встречающийся в коде оператор *if* требует проведения по крайней мере двух тестов (один тест для определения, когда условие вычисляется в истинное значение — *true*, и еще один для определения, когда оно вычисляется в ложное значение — *false*). Оператор *if*, с множеством условий, например:

```
if ( a || b || c )
```

может, по теории, потребовать до восьми тестов — по одному для каждой возможной комбинации значений *a*, *b* и *c*. Добавите к этому управляющие циклы, множество входных параметров, зависимости от внешнего кода, различных аппаратных и программных платформ и т. п., и количество и разновидности тестов нужно будет значительно увеличить.

Любой нетривиальный код, независимо от того, красив он или нет, нуждается не в одном, а в целой *группе* тестов, в которой каждый тест должен быть направлен на проверку специфического аспекта этого кода, подобно тому как разные игроки в спортивной команде отвечают за выполнение различных задач на различных участках игрового поля.

Теперь, когда мы определили, что тесты нужно оценивать в группах, нам нужно понять, какие характеристики могут сделать группу тестов *красивой* — то есть применить к ним редко употребляемое в таких случаях прилагательное.

В общем, основная цель проведения тестов заключается в том, чтобы зародить, укрепить или подтвердить нашу уверенность в правильной и эффективной работе программного кода. Поэтому я считаю, что к наиболее красивым можно отнести те тесты, которые помогают мне максимально укрепить уверенность, что код делает и будет делать то, для чего он предназначен. Поскольку для проверки различных свойств кода нужны разные виды тестов, основной критерий красоты будет изменяться. В данной главе исследуются три направления, позволяющие считать тесты красивыми.

Тесты, красивые своей простотой

Используя всего несколько строк теста, я могу задокументировать и проверить основное поведение испытуемого кода. Автоматически запуская такие тесты с каждой сборкой, я могу убеждаться в том, что по мере развития кода заданное ему поведение сохраняется. В данной главе в качестве примера основных тестов, на которые уходит несколько минут и которые затем продолжают приносить дивиденды на всем жизненном цикле проекта, используется тестовая платформа JUnit.

Тесты, красивые тем, что они открывают пути придания коду большего изящества, удобства в сопровождении и тестировании

Другими словами, это тесты, которые помогают делать код красивее. Процесс написания тестов зачастую помогает разрешить не только логические проблемы, но также и структурные и конструкторские вопросы реализации проекта. В этой главе будет показано, как, создавая тесты, я открыл способ придания программному коду большей надежности, лучшей читаемости и структурированности.

Тесты, красивые широтой своего диапазона и доскональностью

Всестороннее и доскональное тестирование позволяет разработчику приобрести уверенность в том, что код функционирует должным образом, и не только в основных или выбранных, но и во *всех* возможных обстоятельствах. В этой главе будет показано, как я создавал и запускал эту категорию тестов, используя концепцию *теории тестирования*.

Поскольку большинство разработчиков уже знакомы с основными технологиями тестирования, к примеру, с тестами на отсутствие «задымления» и гра ничными испытаниями, я уделю больше времени высокоэффективным разновидностям тестирования и технологиям его проведения, которые довольно редко применяются и обсуждаются.

Изрядно поднадоеvший двоичный поиск

Чтобы продемонстрировать различные виды тестирования, придерживаясь разумных размеров главы, мне нужны примеры, поддающиеся простому описанию и реализуемые в нескольких кодовых строках. В то же время примеры

должны быть достаточно эффективными, предоставляя ряд интересных и проблемных вопросов тестирования. В идеале у примера должна быть богатая предыстория неудачных попыток реализации, демонстрирующая необходимость проведения такого тестирования. И последнее, но не менее важное: было бы не плохо, если этот пример сам по себе можно было рассматривать как красиво исполненный код. Трудно вести речь о красивом коде без упоминания классической книги Иона Бентли (Jon Bentley) «Programming Pearls», выпущенной издательством Addison-Wesley. Перечитывая эту книгу, я и нашел искомый пример красивого кода: это был двоичный поиск.

Напоминаю, что двоичный поиск является простым и эффективным алгоритмом (но, как мы увидим, не столь простым для безошибочной реализации) определения, имеется ли в заранее отсортированном числовом массиве $x[0..n-1]$ искомый элемент t . Если массив содержит t , программа возвращает его позицию в массиве; в противном случае она возвращает -1.

А вот как Ион Бентли описывал этот алгоритм своим студентам:

Двоичный поиск решает задачу, отслеживая диапазон внутри массива, содержащего t (если, конечно, t присутствует в массиве). Сначала этот диапазон равен размеру всего массива. Диапазон сужается путем сравнения среднего элемента массива с t и отбрасывания его ненужной половины. Процесс продолжается до тех пор, пока t не будет найден в массиве, или до тех пор, пока диапазон, в котором он должен был находиться, не станет заведомо пустым.

Затем он добавил:

Большинство программистов думают, что имея в своем распоряжении вышезложенное описание, они смогут с легкостью написать соответствующий код. Но они заблуждаются. Единственный способ убедиться в этом — написать код на основе этого определения прямо сейчас. Попробуйте и посмотрите, что из этого получится.

Я поддерживаю предложение, высказанное Бентли. Если вы никогда не занимались реализацией двоичного поиска или не занимались им в течение нескольких лет, я предлагаю перед тем, как читать дальше, попробовать свои силы; тогда вы сможете лучше оценить все последующее.

Двоичный поиск — превосходный пример, поскольку наряду со своей простотой, он столь же легко и просто может быть неправильно реализован. В своей книге «Programming Pearls» Ион Бентли рассказал, как он в течение нескольких лет обращался с просьбой к сотням профессиональных программистов реализовать двоичный поиск после предоставления им описания его основного алгоритма. Он щедро отпускал им два часа на решение этой задачи и даже разрешал воспользоваться языком высокого уровня по их выбору (включая псевдокод). Как ни удивительно, но только 10 % профессиональных программистов справлялись с решением задачи двоичного поиска безошибочно.

Не менее удивительный факт привел в своей книге «Sorting and Searching»¹, Дональд Кнут (Donald Knuth): хотя впервые идея двоичного поиска была опуб-

¹ «The Art of Computer Programming, Vol. 3: Sorting and Searching». Donald Knuth, Second Edition, Addison-Wesley, 1998.

ликована в 1946 году, понадобилось более 12 лет для того, чтобы был опубликован первый безошибочный код двоичного поиска.

Но наиболее удивительным является то, что, оказывается, в официальном и доказавшем свою состоятельность алгоритме Иона Бентли, который (смею предположить) был реализован и применен на практике не одну тысячу раз, скрывалась проблема, которая проявила себя при достаточно больших массивах и при реализации алгоритма на языках, имеющих арифметику заданной точности.

В Java этот изъян проявил себя вызовом исключения `ArrayIndexOutOfBoundsException`, тогда как в Си получалось, что индекс массива выходил за пределы с непредсказуемыми результатами. О последней из этих ошибок можно прочитать на блоге Джошуа Блоха (Joshua Bloch): <http://googleresearch.blogspot.com/2006/06/extraextra-read-all-about-it-nearly.html>.

А вот как выглядит реализация на языке Java, в которой не удалось избежать этой досадной ошибки:

```
public static int buggyBinarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

Ошибка кроется в следующей строке:

```
int mid = (low + high) / 2;
```

Если сумма значений `low` и `high` больше, чем значение `Integer.MAX_VALUE` (которое в Java составляет $2^{31} - 1$), то переполнение приводит к появлению отрицательного числа, которое, разумеется, остается отрицательным и после деления на два! Рекомендуемое решение заключается в изменении вычисления средней точки, с тем чтобы избежать целочисленного переполнения. Один из способов реализации состоит в замене сложения вычитанием:

```
int mid = low + ((high - low) / 2);
```

Или же если вы желаете блеснуть знаниями операторов разрядного сдвига, то блог (и официальный веб-сайт сообщений об ошибках компании *Sun Microsystems*¹) предлагает воспользоваться беззнаковым разрядным сдвигом, который,

¹ http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5045582.

наверное, работает быстрее, но может быть недопонят большинством разработчиков Java-программ (включая и меня):

```
int mid = (low + high) >> 1;
```

Учитывая, насколько проста сама идея двоичного поиска, количество людей, работавших над ней все эти годы, и затраченные ими коллективные интеллектуальные усилия, мы имеем превосходный пример, показывающий, почему даже простейший код нуждается в тестировании, и таких примеров великое множество. Джошуа Блок довольно красочно написал об этом в своем веб-блоге, посвященном рассматриваемой ошибке:

Главный урок, который я вынес из рассмотрения этой ошибки, заключается в смирении: довольно трудно избежать ошибок при написании даже самого маленького фрагмента кода, а весь наш мир управляет с помощью куда более значительных и сложных кодовых фрагментов.

Я хочу протестировать следующий вариант реализации двоичного поиска. Теоретически, исправления, внесенные в способ вычисления `mid`, должны устранить последнюю ошибку в той самой, вызывающей чувство досады части кода, которая ускользала от внимания лучших программистов в течение нескольких десятилетий:

```
public static int binarySearch(int[] a, int target) {  
    int low = 0;  
    int high = a.length - 1;  
  
    while (low <= high) {  
        int mid = (low + high) >> 1;  
        int midVal = a[mid];  
  
        if (midVal < target)  
            low = mid + 1;  
        else if (midVal > target)  
            high = mid - 1;  
        else  
            return mid;  
    }  
    return -1;  
}
```

Эта версия `binarySearch` выглядит безупречной, но в ней по-прежнему могут скрываться проблемные места. Возможно, речь будет идти не об ошибках, а о тех местах, которые можно и нужно изменить. Изменения сделают код не только более надежным, но и легче читаемым, сопровождаемым и тестируемым. Давайте посмотрим, можно ли в процессе тестирования обнаружить некоторые интересные и неожиданные возможности для усовершенствования кода.

Знакомство с JUnit

Говоря о красивых тестах, трудно не вспомнить о тестовой платформе JUnit. Поскольку я пользуюсь Java, мне проще всего было принять решение о построении красивых тестов на основе JUnit. Но перед осуществлением своих

замыслов, на тот случай, если вы еще не знакомы с JUnit, позвольте сказать о ней несколько слов.

JUnit – это плод творческих изысканий Кента Бека (Kent Beck) и Эрика Гамма (Erich Gamma), создавших ее, чтобы помочь Java-разработчикам создавать и запускать автоматизированные, самостоятельно осуществляющие проверку тесты. При ее создании вынашивалась простая, но амбициозная цель – облегчить разработчикам программного обеспечения то, что они должны делать всегда: тестиирование их собственного кода.

К сожалению, для того чтобы большинство разработчиков заразились идеей тестиирования (то есть провели эксперименты по тестиированию в процессе разработки и пришли к решению сделать его регулярной и существенной составляющей своей практической деятельности), нужно приложить еще немало усилий. Тем не менее со временем своего представления JUnit (во многом благодаря экстремальному программированию и другим методологиям ускоренной разработки, которые предусматривают безоговорочное подключение разработчика к тестиированию) привлекла к написанию тестов больше программистов, чем какая-нибудь другая разработка¹. Мартин Фоулер (Martin Fowler) резюмировал влияние JUnit следующим образом: «Ничто в сфере разработки программного обеспечения не давало столь ощутимой отдачи при столь малом количестве строк кода».

В JUnit простота закладывалась намеренно. Эта платформа так же проста в изучении, как и в использовании. Это свойство было ключевым критерием ее разработки. Кент Бек и Эрик Гамма приложили огромные усилия, чтобы обеспечить легкость освоения и использования JUnit, позволяющую программистам перейти к ее практическому применению. По их собственным словам:

Главная цель состояла в том, чтобы написать платформу, которая дала бы нам проблески надежды на то, что разработчики действительно станут писать тесты. В платформе должны использоваться уже знакомые инструменты, чтобы на изучение нововведений не приходилось тратить много усилий. Объем работы должен в точности соответствовать потребностям написания нового теста. Все дублирующие моменты должны быть исключены².

Официальная документация, позволяющая приступить к работе с JUnit (так называемый сборник рецептов – «JUnit Cookbook»), помещается менее чем на двух страницах: <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>.

Ключевые извлечения из сборника рецептов (из версии JUnit 4.x) выглядят следующим образом.

При желании что-нибудь протестировать, сделайте следующее:

¹ Другим свидетельством успеха и степени воздействия JUnit является существование в настоящее время разработанных под ее влиянием подобных платформ для наиболее современных языков программирования, а также многих JUnit-расширений.

² «JUnit: A Cook's Tour», Кент Бек (Kent Beck) и Эрик Гамма (Erich Gamma): <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>.

1. метод, предназначенный для тестирования, промаркируйте аннотацией `@org.junit.Test`;
2. когда нужно проверить значение, импортируйте статически `org.junit.Assert.*`, вызовите метод `assertTrue()` и передайте ему логическое значение, которое примет истинное значение, если тест пройдет успешно.

Например, чтобы проверить, что сумма двух значений Money, относящихся к одной и той же валюте, соответствует ожидаемой величине, напишите:

```
@Test
public void simpleAdd() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");
    Money expected= new Money(26, "CHF");
    Money result= m12CHF.add(m14CHF);
    assertTrue(expected.equals(result));
}
```

Если вам в какой-то мере знаком язык Java, то для начала вам хватит лишь этих двух инструкций и простого примера. Также это все, что вам понадобится, чтобы разобраться в тех тестах, которые я буду создавать. Красивый пример, не правда ли? А теперь примемся за работу.

Подробный разбор двоичного поиска

Памятуя о предыстории двоичного поиска, я не стану впадать в заблуждение из-за его очевидной простоты или из-за наглядности внесенного исправления, особенно потому, что в любом другом коде я никогда не пользовался оператором беззнакового разрядного сдвига (то есть `>>>`). Я собираюсь протестировать эту *исправленную* версию двоичного поиска, как будто никогда раньше о ней не слышал и никогда ею не пользовался. Я не собираюсь верить кому-то на слово или доверять каким-то тестам или доказательствам, пока этот вариант не будет работать по-настоящему. Я должен удостовериться в его работоспособности, проводя свое собственное тестирование. Мне нужно разобраться в нем *досконально*.

Изначально стратегия моего тестирования (или подборка тестов) выглядит следующим образом.

- Начну с *тестов на задымление*.
- Добавлю к ним ряд *тестов на граничные значения*.
- Продолжу работу со всесторонними и доскональными видами тестов.
- И в заключение добавлю ко всему этому несколько *тестов на производительность*.

Тестирование редко проходит гладко. Вместо того чтобы показать окончательно сложившийся набор тестов, я хочу провести вас через весь ход моих размышлений в процессе работы с тестами.

Задымление разрешается (и приветствуется)

Начнем с тестов на задымление. Они предназначены для того, чтобы убедиться в общей работоспособности кода. Это первый защитный рубеж, и первые тесты, которые должны быть написаны, поскольку если созданный код не пройдет тесты на задымление, все последующее тестирование будет пустой тратой времени. Зачастую я пишу тесты на задымление еще до написания самого программного кода; такой подход называется *разработкой под тестированием* (test-driven development, или TDD).

А вот как выглядит мой тест на задымление для двоичного тестирования:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BinarySearchSmokeTest {

    @Test
    public void smokeTestsForBinarySearch() {
        int[] arrayWith42 = new int[] { 1, 4, 42, 55, 67, 87, 100, 245 };
        assertEquals(2, Util.binarySearch(arrayWith42, 42));
        assertEquals(-1, Util.binarySearch(arrayWith42, 43));
    }
}
```

Можно сказать, что этот тест *по-настоящему* основной. Сам по себе он не отличается полноценностью, но красив уже только потому, что является очень быстрым и эффективным первым шагом по направлению к более доскональным тестам. Поскольку этот тест на задымление выполняется очень быстро (на моей системе — менее чем за одну сотую секунды), может быть задан вопрос, почему я не включил в него еще несколько тестов. Ответ является частью той красоты, которая присуща тестам на задымление — они могут продолжать приносить пользу после того, как основная часть разработки уже будет выполнена. Чтобы подтвердить мою уверенность в коде — назовем ее «поддержкой уверенности», — я люблю собирать все тесты на задымление в набор программ, который запускаю для каждой новой конструкции (что может случаться десятки раз за день), и я хочу, чтобы эта подборка тестов на задымление работала быстро, в идеале одну-две минуты. Если вы имеете дело с тысячами классов и тысячами тестов на задымление, важно каждый из них свести к необходимому минимуму.

Проталкивание через границы

Судя по названию, тесты на граничные значения создаются для исследования и проверки того, что получится, когда код столкнется с экстремальными или крайними случаями. В отношении двоичного поиска есть два параметра — мас-

сив и искомое значение. Нужно подумать, какие крайние случаи могут быть для каждого из этих параметров¹.

Самая первая мысль о крайних случаях касается размера того массива, в котором производится поиск. Начну со следующих граничных тестов:

```
int[] testArray;
```

```
@Test
public void searchEmptyArray( ) {
    testArray = new int[] {};
    assertEquals(-1, Util.binarySearch(testArray, 42));
}

@Test
public void searchArrayOfSizeOne( ) {
    testArray = new int[] { 42 };
    assertEquals(0, Util.binarySearch(testArray, 42));
    assertEquals(-1, Util.binarySearch(testArray, 43));
}
```

Понятно, что хорошим граничным случаем будет пустой массив и массив единичного размера, как наименьший непустой массив. Оба эти теста красивы тем, что укрепляют мою уверенность, что на нижних границах размеров массива все происходит должным образом.

Но я также хочу протестировать поиск в массиве очень большого размера, и здесь ситуация приобретает особый интерес (учитывая вышеизложенные сведения о том, что ошибка проявляется только при работе с массивами, состоящими из более чем миллиарда элементов).

Первой моей мыслью было создать достаточно большой массив, чтобы убедиться в том, что ошибка целочисленного переполнения исправлена, но я тут же обнаружил проблему, связанную с ограничениями своих возможностей по тестированию: мой ноутбук не обладал достаточным объемом ресурсов для создания в памяти столь большого массива. Но я знал, что существуют системы, в которых имеются многие гигабайты памяти, позволяющие хранить столь большие массивы. Я все же хотел тем или иным способом убедиться, что целочисленная переменная `mid` в подобных случаях не переполняется.

И что для этого нужно было сделать?

Я знаю, что к тому времени, когда я закончу разработку некоторых других задуманных мною тестов, их будет достаточно, чтобы удостовериться в работоспособности основного алгоритма и его реализации *при условии, что середина вычислена правильно, и ей не присваивается отрицательное значение в результате переполнения*.

К возможной стратегии тестирования огромных массивов меня привели следующие рассуждения.

¹ В спецификации двоичного поиска говорится, что перед его вызовом массив должен быть отсортирован. А если он не отсортирован, то будут получены неопределенные результаты. Также предполагается, что использование в параметре вместо массива значения `null` приведет к вызову исключения `NullPointerException`. Поскольку большинство читателей уже знакомы с основами технологии граничного тестирования, я собираюсь опустить некоторые, самые очевидные из этих тестов.

1. Я не могу проводить непосредственное тестирование функции `binarySearch` с массивами достаточно большого размера, чтобы проверить, что ошибка переполнения при вычислении значения переменной `mid` больше не возникает.
2. Тем не менее я могу написать достаточно много тестов, которые дадут мне уверенность, что моя реализация функции `binarySearch` с массивами меньшего размера работает безошибочно.
3. Я также могу протестировать способ вычисления значения `mid` при использовании очень больших значений без привлечения массивов.
4. И если, проводя эти тесты, я смогу получить достаточную степень уверенности в следующем:
 - моя реализация базового алгоритма `binarySearch` не имеет дефектов, пока `mid` вычисляется правильно, и
 - при безошибочном способе вычисления середины я могу быть уверен, что `binarySearch` будет правильно работать и с очень большими массивами.

Итак, не столь очевидная, но красавая стратегия тестирования заключается в изоляции и независимом тестировании того самого досадного, склонного к переполнению вычисления.

Один из возможных вариантов заключается в создании нового метода:

```
static int calculateMidpoint(int low, int high) {
    return (low + high) >>> 1;
}
```

с изменением следующей строки кода с

```
int mid = (low + high) >>> 1;
```

на

```
int mid = calculateMidpoint(low, high);
```

и последующей проверки самого узкого места во внешнем методе `calculateMidpoint`, чтобы убедиться в том, что все делается правильно.

Я уже могу услышать, как некоторые из моих читателей возмущаются добавлением в алгоритм, предназначенный для достижения максимального быстродействия, дополнительного вызова метода. Но эти возмущения абсолютно напрасны. Я полагаю, что внесенные в код изменения не только приемлемы, но и правильны по следующим соображениям.

1. В наше время я могу довериться оптимизационным возможностям компилятора, позволяющим ему все сделать правильно и встроить для меня этот метод, не допустив снижения производительности.
2. Изменения делают код более читаемым. Я обменялся мнениями с несколькими другими программистами, работающими на Java, и большинство из них оказались незнакомы с оператором беззнакового разрядного сдвига или не были на 100 % уверены в том, как он работает. Для них вариант `calculateMidpoint(low, high)` представлялся более наглядным, чем `(low + high) >>> 1`.
3. Изменения сделали код более приспособленным к тестированию.

Это действительно неплохой пример того, как именно те усилия, которые направлены на создание теста для вашего кода, приведут к повышению его

конструктивных качеств или удобочитаемости. Другими словами, тестирование способствует приданию нашему коду дополнительной красоты.

А вот так выглядит пример граничного теста для вновь созданного метода `calculateMidpoint`:

```
@Test
public void calculateMidpointWithBoundaryValues( ) {
    assertEquals(0, calculateMidpoint(0, 1));
    assertEquals(1, calculateMidpoint(0, 2));
    assertEquals(1200000000, calculateMidpoint(1100000000, 1300000000));
    assertEquals(Integer.MAX_VALUE - 2,
        calculateMidpoint(Integer.MAX_VALUE - 2, Integer.MAX_VALUE - 1));
    assertEquals(Integer.MAX_VALUE - 1,
        calculateMidpoint(Integer.MAX_VALUE - 1, Integer.MAX_VALUE));
}
```

Я запустил тесты, и они были пройдены. Ну хорошо, а теперь, когда я уверен, что вычисление `mid` в пределах рабочего диапазона размеров массива с применением непривычного для меня оператора происходит, как и ожидалось, я хочу заняться реализацией двоичного поиска.

Другая подборка граничных случаев предназначена для работы с позицией искомого числа. Я думаю, что существуют три вполне очевидных граничных значения для позиции целевого элемента: первый элемент списка, последний элемент списка и элемент, находящийся точно посередине списка. Для проверки этих случаев я написал довольно простой тест:

```
@Test
public void testBoundaryCasesForItemLocation( ) {
    testArray = new int[] { -324, -3, -1, 0, 42, 99, 101 };
    assertEquals(0, Util.binarySearch(testArray, -324)); // первая позиция
    assertEquals(3, Util.binarySearch(testArray, 0)); // средняя позиция
    assertEquals(6, Util.binarySearch(testArray, 101)); // последняя позиция
}
```

Заметьте, что в этом teste я использовал несколько отрицательных чисел и нуль, как для массива, так и для искомого числа. Так уж случилось, что читая уже написанные тесты, я увидел, что в них использовались только положительные числа. Поскольку это не являлось частью спецификации, мне нужно было представить в своих тестах отрицательные числа и нуль. Это обстоятельство натолкнуло меня на следующую мудрую мысль, относящуюся к тестированию:

Лучший способ придумать побольше случаев для тестирования – это приступить к записи *каких-нибудь* случаев.

Теперь, после того как я задумался о необходимости присутствия в teste положительных, отрицательных чисел и нуля, я понял, что неплохо было бы занять парочку тестов, в которых использовались бы максимальные и минимальные целочисленные значения.

```
public void testForMinAndMaxInteger( ) {
    testArray = new int[] {
        Integer.MIN_VALUE, -324, -3, -1, 0, 42, 99, 101, Integer.MAX_VALUE
    };
    assertEquals(0, Util.binarySearch(testArray, Integer.MIN_VALUE));
```

```
assertEquals(8, Util.binarySearch(testArray, Integer.MAX_VALUE));  
}
```

Пока все задуманные граничные случаи были пройдены, и я начал приобретать уверенность. Но когда я подумал о тех 90 % профессиональных программистах из числа тех, кому Ион Бентли предложил реализовать программу двоичного поиска, которые, думая, что справились с задачей, оказались неправы, моя уверенность несколько поколебалась. Все ли я предусмотрел во входных данных? Ведь вплоть до этого последнего теста я же не подумал об отрицательных числах и нуле. Какие еще неоправданные предположения я смог допустить? Поскольку я создавал тесты самостоятельно, то мог подсознательно создать работоспособные случаи и упустить ряд других, при которых они потерпели бы неудачу.

Это общеизвестная проблема программистов, тестирующих собственный код. Если они не смогли продумать какие-то сценарии развития событий при создании кода, то, скорее всего, они не смогут их продумать и когда переключатся на попытки *нарушить работу* этого кода. По-настоящему красивое тестирование требует от разработчика особых усилий, нетривиального мышления, исследования, казалось бы, самых невероятных сценариев, поиска слабых мест и попыток разрушить свое творение.

Итак, что же я мог упустить? Что-то в моих тестах на задумленность и граничных испытаниях не чувствуется достаточности. Неужели мои тесты настолько показательны, что я с помощью некой формы индуктивного мышления¹ смогу утверждать, что код в любых случаях будет работоспособен? Слова Джошуа Блоча отдавались эхом в моем сознании: «...трудно избежать ошибок при написании даже самого маленького фрагмента кода».

Какие же разновидности тестов возбудят во мне достаточное чувство уверенности, что моя реализация будет работать правильно с любыми входными данными, а не только с теми, которые я создал своими собственными руками?

Элемент случайности в тестировании

До сих пор я писал традиционные тесты, хорошо зарекомендовавшие себя на практике. При этом для тестирования поиска на ожидаемое поведение в том или ином случае я использовал некие конкретные примеры. Все эти тесты были пройдены, и я обрел некоторую уверенность в своем коде. Но я также осознал, что мои тесты слишком специфичны и охватывают лишь небольшую часть диапазона всех возможных входных данных. Чего бы я хотел и что бы помогло мне спокойно спать по ночам, зная, что мой код был протестирован во всем диапазоне, так это способ тестирования с использованием куда более широкого набора входных значений. Чтобы реализовать это желание, мне нужны две вещи:

¹ Под индуктивным мышлением я понимаю извлечение общих принципов из конкретных фактов или случаев.

- Способ генерирования большого и разнообразного набора входных значений;
- Ряд обобщенных утверждений, которые будут верны для любых входных данных.

Начнем с первой потребности.

Мне нужно найти способ генерирования массивов целых чисел любых форм и размеров. Но при этом нужно соблюсти единственное требование: получаемый массив должен быть отсортирован, поскольку в этом заключается предварительное условие проведения двоичного поиска. Все остальное, кроме этого, вполне допустимо. Вот как выглядит самая первая реализация генератора¹:

```
public int[] generateRandomSortedArray(int maxArraySize, int maxValue) {  
    int arraySize = 1 + rand.nextInt(maxArraySize);  
    int[] randomArray = new int[arraySize];  
    for (int i = 0; i < arraySize; i++) {  
        randomArray[i] = rand.nextInt(maxValue);  
    }  
    Arrays.sort(randomArray);  
    return randomArray;  
}
```

В этом генераторе я воспользовался генератором случайных чисел из утилит, имеющихся в Java, а также утилитами Arrays. Последние содержат в себе ту же ошибку двоичного поиска, о которой упоминал Джошуа Блоч, но в той версии Java, которую я использовал, она уже устранена. Поскольку для удовлетворения своих потребностей я уже использовал случай пустых массивов в других своих тестах, здесь я воспользовался минимальным размером массива, равным единице. В генераторе используются входные параметры, поскольку у меня в процессе работы может появиться желание создать различные наборы тестов: с небольшими массивами, содержащими большие числа, и с большими массивами, содержащими маленькие числа, и т. д.

Теперь мне нужно определить некоторые общие положения в требуемом поведении двоичного поиска, которые можно было бы выразить в виде утверждений. Под термином «общие» я подразумеваю утверждения, которые должны быть верны для любого входного массива и любого искомого значения. Мои коллеги, Марат Бошерницын (Marat Boshernitsan) и Дэвид Сафф (David Saff), называют это *предположениями*. Идея заключается в том, что у нас имеется теория о том, как код должен себя вести, и чем больше мы подвергаем теорию проверке, тем больше в нас крепнет уверенность, что наши теоретические выкладки на самом деле верны. В следующем примере я собираюсь применить весьма упрощенную версию теоретических изысканий Саффа и Бошерницына.

Попытаемся придумать некие теоретические выкладки для binarySearch. Вот что у нас может получиться.

¹ Я сказал, что это самая первая реализация, поскольку быстро понял, что массив нужно заполнять не только положительными, но и отрицательными числами, и внес в генератор соответствующие изменения.

Для всех экземпляров `testArray` и `target`, где `testArray` — это отсортированный массив, содержащий целые числа, не принимающие значение `null`, а `target` — это целое число, следующие утверждения в `binarySearch` должны быть истинными.

Теория № 1: если `binarySearch(testArray, target)` возвращает значение `-1`, значит, `testArray` не содержит `target`.

Теория № 2: если `binarySearch(testArray, target)` возвращает значение `n` и `n` больше или равно нулю, значит, `testArray` содержит `target` в позиции `n`.

Мой код для проверки этих двух теорий имеет следующий вид:

```
public class BinarySearchTestTheories {

    Random rand;

    @Before
    public void initialize() {
        rand = new Random();
    }

    @Test
    public void testTheories() {

        int maxArraySize = 1000;
        int maxValue = 1000;
        int experiments = 1000;

        int[] testArray;
        int target;
        int returnValue;

        while (experiments-- > 0) {
            testArray = generateRandomSortedArray(maxArraySize, maxValue);
            if (rand.nextBoolean()) {
                target = testArray[rand.nextInt(testArray.length)];
            } else {
                target = rand.nextInt();
            }
            returnValue = Util.binarySearch(testArray, target);
            assertTheory1(testArray, target, returnValue);
            assertTheory2(testArray, target, returnValue);
        }
    }

    public void assertTheory1(int[] testArray, int target, int returnValue) {
        if (returnValue == -1)
            assertFalse(arrayContainsTarget(testArray, target));
    }

    public void assertTheory2(int[] testArray, int target, int returnValue) {
        if (returnValue >= 0)
            assertEquals(target, testArray[returnValue]);
    }
}
```

```
public boolean arrayContainsTarget(int[] testArray, int target) {  
    for (int i = 0; i < testArray.length; i++)  
        if (testArray[i] == target)  
            return true;  
    return false;  
}
```

В главном тестовом методе `testTheories` я решаю, сколько проб хочу произвести в подтверждение этих теорий, и использую это число в качестве счетчика цикла. Внутри цикла уже написанный мною генератор случайного массива выдает мне отсортированный массив. Я хочу протестировать как удачные, так и неудачные поиски, поэтому опять использую принадлежащий Java генератор случайных чисел «для подбрасывания монетки» (при помощи кода `rand.nextIntBoolean()`). На основе виртуального подбрасывания монетки я решаю, собираюсь ли я подобрать искомое значение из тех, что *заведомо* находятся в массиве, или взять такое значение, которого там, скорее всего, нет. И в заключение я вызываю `binarySearch`, сохраняю возвращаемое значение и задействую методы для тех теорий, которые разработаны на данный момент.

Заметьте, что в целях осуществления тестов для моих теорий мне понадобилось написать тестовый вспомогательный метод `arrayContainsTarget`, который дал мне альтернативный способ проверки, содержит ли `testArray` искомый элемент. Для таких тестов в этом нет ничего необычного. Даже если реализация этого вспомогательного метода обеспечивает функциональность, сопоставимую с `binarySearch`, она намного проще (хотя и намного медленнее) реализации поиска. Я уверен, что вспомогательный метод работает правильно, поэтому могу использовать его для тестирования той реализации, в правильности которой уверен намного меньше.

Начинаю с запуска 1000 экспериментов над массивами размером до 1000 элементов. Тесты занимают долю секунды, и все проходит нормально. Хорошо. Теперь настало время немного расширить область исследований (не забывайте, что тестирование относится к исследовательской деятельности).

Изменяя условия эксперимента, я увеличиваю значение `maxArraySize` сначала до 10 000, а потом и до 100 000. Теперь тесты занимают почти минуту и полностью задействуют мой процессор. Я чувствую, что код получает по-настоящему хорошую нагрузку.

Моя уверенность возрастает, но я убежден в том, что *если все тесты прошли успешно, то вполне возможно, что они недостаточно хороши*. Какие другие свойства мне нужно протестировать, располагая уже созданной структурой?

Я немного подумал и заметил, что обе мои теории укладываются в одну форму:

Если что-либо истинно в отношении возвращаемого `binarySearch` значения, то должно быть истинным и что-нибудь еще, относящееся к `testArray` и `target`.

Другими словами, логика укладывается в форму из p вытекает q (или $p \rightarrow q$, используя логическую нотацию), а значит, я протестировал только половину из

того, что должно быть протестировано. Мне также нужны тесты, имеющие форму $q \rightarrow p^1$:

Если что-либо истинно в отношении `testArray` и `target`, то должно быть истинным и что-нибудь еще, относящееся к возвращаемому значению.

Немного сложно, но важно, поэтому я кое-что должен пояснить. Тесты, предназначенные для подтверждения теории № 1, проверяют, что возвращаемое значение равно `-1`, то есть искомый элемент в массиве отсутствует. Но они не проверяют того, что при отсутствии в массиве искомого элемента возвращаемое значение равно `-1`. Другими словами: *если бы у меня была для тестирования только лишь эта теория*, реализация метода, возвращающая время от времени `-1`, но не во *всех* случаях, когда она должна была это делать, все равно прошла бы все мои тесты. Аналогичная проблема существует и для теории № 2.

Я могу это продемонстрировать при помощи *мутационного тестирования*, техники тестирования тестов, изобретенной Джейфом Оффутом (Jeff Offut). Основная идея состоит в том, чтобы внести в тестируемый код заведомо известные ошибки. Если тесты по-прежнему проходят, несмотря на ошибку в коде, значит, эти тесты не обладают требуемой доскональностью.

Проведем мутацию `binarySearch`, воспользовавшись несколько радикальным и произвольным способом. Я попытаюсь сделать следующее: если значение `target` превышает число `424242` и `target` не содержится в массиве, вместо возвращения `-1`, я собираюсь вернуть `-42`. Как насчет программного вандализма? Обратите внимание на нижнюю часть следующего кода:

```
public static int binarySearch(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return mid;
    }
    if (target <= 424242)
        return -1;
```

¹ Разумеется, и p , и q , и они оба могут быть отрицаниями (то есть $\neg p \rightarrow \neg q$, or $p \rightarrow \neg q$). Я совершенно произвольно выбрал p и q в качестве заместителей для любых утверждений, касающихся, соответственно, возвращаемого значения и параметра массива. Тут важно осознать, что в процессе программирования вы обычно рассуждаете в понятиях $p \rightarrow q$ (если p будет истиной, то q должно произойти — это так называемый удачный исход, являющийся нормой для наиболее общих случаев использования кода). Но при тестировании вы должны себя заставить мыслить и в обратном направлении ($q \rightarrow ?$, или если q будет истинным, то должно ли быть истинным p ?), и в отрицательном смысле (если p не будет истиной [то есть $\neg p$], то должно ли быть истинным q ?).

```

    else
        return -42;
}

```

Надеюсь, вы согласитесь с тем, что это довольно большая мутация: код возвращает не ожидавшееся и не определенное техническими условиями значение, если искомое число больше, чем 424242, и оно не содержится в массиве. И все же все созданные нами на данный момент тесты проходили «на ура». Чтобы усилить тесты и отловить подобную разновидность мутаций, нам определенно не хватает еще нескольких теорий.

Теория № 3: если `testArray` не содержит искомого значения — `target`, то метод должен возвращать `-1`.

Теория № 4: если `testArray` содержит `target` в позиции `n`, то `binarySearch(testArray, target)` должен возвращать `n`.

Эти теории тестируются следующим кодом:

```

public void assertTheory3(int[] testArray, int target, int returnValue) {
    if (!arrayContainsTarget(testArray, target))
        assertEquals(-1, returnValue);
}

public void assertTheory4(int[] testArray, int target, int returnValue) {
    assertEquals(getTargetPosition(testArray, target), returnValue);
}

public int getTargetPosition(int[] testArray, int target) {
    for (int i = 0; i < testArray.length; i++)
        if (testArray[i] == target)
            return i;
    return -1;
}

```

Обратите внимание, что мне понадобилось создать еще один вспомогательный метод, `getTargetPosition`, который делает абсолютно то же самое, что и `binarySearch` (но я уверен, что он работает правильно, имея огромный недостаток, потому что ему требуется провести до n сравнений вместо $\log_2 n$). Поскольку метод `getTargetPosition` весьма похож на `arrayContainsTarget`, а дублировать код нехорошо, я переписал последний метод следующим образом:

```

public boolean arrayContainsTarget(int[] testArray, int target) {
    return getTargetPosition(testArray, target) >= 0;
}

```

Я снова запустил эти тесты, воспользовавшись своим генератором случайных массивов, и теперь мутация `return -42` была сразу же отловлена. Что ж, не плохо, это позволяет укрепить мою уверенность. Я убрал преднамеренно внесенную ошибку и снова запустил тест. Я ждал, что он будет пройден, но этого не произошло. Не были пройдены тесты, касающиеся теории № 4. JUnit не закончил работу, выдав следующее сообщение:

`expected:<n> but was:<n + 1>`

В теории № 4 говорится о том, что если `testArray` содержит `target` в позиции `n`, то `binarySearch(testArray, target)` должен возвращать `n`.

Значит, иногда процедура поиска возвращает местоположение, отличающееся от истинного на единицу. Почему так происходит?

Мне требуются дополнительные сведения. Утверждение, имеющееся в JUnit, способно в качестве первого параметра воспринимать сообщение типа String, поэтому я изменю отвечающий за проверку теории № 4 метод `assertEqual`, включив в него текст, который даст мне дополнительные сведения в том случае, если тест не будет пройден:

```
public void assertTheory4(int[] testArray, int target, int returnValue) {
    String testDataInfo = "Theory 4 - Array=" +
        printArray(testArray)
        + " target="
        + target;
    assertEquals(testDataInfo, getTargetPosition(testArray, target),
        returnValue);
}
```

Теперь, как только теория № 4 не получила своего подтверждения, JUnit покажет мне содержимое массива и искомое значение. Я запустил тест еще раз (с небольшими значениями `maxArrayList` и `maxValue`, чтобы выводимая информация легче читалась) и получил следующий результат:

```
java.lang.AssertionError: Theory 4 - Array=[2. 11. 36. 66. 104. 108. 108.
108. 122. 155. 159. 161. 191] target=108 expected:<5> but was:<6>
```

Я понял, что получилось. Теория № 4 не берет в расчет продублированные значения, а я об этом не подумал. В массиве содержатся три числа 108. Похоже, мне нужно выработать технические условия, учитывающие продублированные числа, и внести изменения либо в код, либо в мою теорию и тесты. Но я оставлю это в качестве упражнения для своих читателей (мне всегда хотелось это сказать!), поскольку я уже выхожу за рамки отведенного мне в этой книге пространства, но прежде чем закончить эту главу, я хочу сказать еще несколько слов о тестах производительности.

Беспокойства о производительности

Отработанные нами тесты, основанные на теориях, позволили закинуть довольно прочную сеть на реализацию процедуры поиска. Теперь, пожалуй, будет трудно пройти эти тесты, оставив какие-нибудь изъяны в реализации. И все же мы кое-что проглядели. Все наши тесты хороши для поиска, но все, что мы тестировали, относилось конкретно к *двоичному* поиску. А нам нужна подборка тестов, не имеющих к нему *никакого* отношения. Нам нужно установить, действительно ли количество сравнений, производимых реализованной нами процедурой, соответствует ожидаемому максимуму, составляющему $\log_2 n$ сравнений. Так как же нам с этим справиться?

Первое, о чём я подумал, — это воспользоваться системным таймером, но сразу же отверг эту идею, поскольку имеющийся в моем распоряжении таймер не обладал достаточной разрешающей способностью для этих испытаний (двоичный поиск проходит молниеносно), и я практически не могу проконтролировать ту среду, в которой он выполняется. Поэтому я воспользовался другой

уловкой из арсенала разработчиков тестов: я создал другую реализацию процедуры `binarySearch` под названием `binarySearchComparisonsCount`. В этой версии кода используется та же самая логика, что и в оригинале, но в ней ведется подсчет сравнений и возвращается их количество вместо возвращения `-1` или позиции искомого числа¹. Вот как этот код выглядит:

```
public static int binarySearchComparisonCount(int[] a, int target) {
    int low = 0;
    int high = a.length - 1;

    int comparisonCount = 0;

    while (low <= high) {

        comparisonCount++;

        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < target)
            low = mid + 1;
        else if (midVal > target)
            high = mid - 1;
        else
            return comparisonCount;
    }
    return comparisonCount;
}
```

Затем на основе этого кода я создам еще одну теорию.

Теория № 5: если размер `testArray` равняется n , то метод `binarySearchComparisonCount(testArray, target)` должен вернуть число меньше или равное $1 + \log_2 n$.

А вот как выглядит код для проверки этой теории:

```
public void assertTheory5(int[] testArray, int target) {
    int numberOfComparisons =
        Util.binarySearchComparisonCount(testArray, target);
    assertTrue(numberOfComparisons <= 1 + log2(testArray.length));
}
```

Я добавил эту последнюю теорию к уже существующему списку, содержащемуся в методе `testTheories`, который приобрел следующий вид:

```
...
while (experiments-- > 0) {
    testArray = generateRandomSortedArray();
    if (rand.nextInt() % 2 == 0) {
```

¹ Вместо модификации `binarySearch` для получения показаний счетчика сравнений будет лучше, более понятно и более правильно с точки зрения объектно-ориентированного конструирования (предложенного Дэвидом Саффом) создание класса `CountingComparator`, принадлежащего имеющемуся в Java генерализованному интерфейсу `Comparator`, и модификации `binarySearch` для получения экземпляра этого класса в качестве третьего параметра. Тем самым `binarySearch` будет приспособлен к работе с типами данных, отличающимися от целых чисел.

```
target = testArray[rand.nextInt(testArray.length)];
} else {
    target = rand.nextInt( );
}
returnValue = Util.binarySearch(testArray, target);
assertTheory1(testArray, target, returnValue);
assertTheory2(testArray, target, returnValue);
assertTheory3(testArray, target, returnValue);
assertTheory4(testArray, target, returnValue);
assertTheory5(testArray, target);
}
```

...

Я запустил несколько тестов, устанавливая разные значения для `maxArraySize`, и понял, что положения теории № 5, похоже, соблюдаются.

Поскольку близится полдень, я установил для количества экспериментов — `experiments` значение 1 000 000, и пошел обедать, пока мой компьютер изо всех сил проводил тестирование каждой теории по миллиону раз.

Когда я вернулся, то увидел, что все тесты пройдены. Может быть, мне захочется протестировать еще парочку вещей, но я достиг большого прогресса в укреплении *своей* уверенности в реализации процедуры `binarySearch`. Поскольку у всех разработчиков разное образование, стиль работы и уровень опыта, то вы можете сконцентрировать свое внимание на других участках кода. К примеру, у разработчика, уже знакомого с оператором беззнакового сдвига, не будет таких же потребностей в его тестировании, какие были у меня.

В этом разделе я хотел заинтересовать вас тестированием производительности и показать, как можно получить сведения и уверенность в производительности вашего кода, сочетая программное оснащение с теориями тестирования. Я настоятельно вам рекомендую изучить главу 3, в которой Ион Бентли уделяет этой важной теме достойное внимание и дает ей красивое толкование.

Вывод

В этой главе мы увидели, что даже лучшие разработчики и самый красивый код могут выиграть от тестирования. Мы также увидели, что создание кода для тестирования может быть таким же творческим и захватывающим занятием, как и написание целевого кода. Будем надеяться, мне удалось вам показать, что по крайней мере по трем различным показателям можно считать красивыми и сами тесты.

Некоторые тесты красивы своей простотой и эффективностью. Используя всего лишь несколько строк JUnit-кода, запускаемых автоматически с каждой новой сборкой, вы сможете задокументировать предопределенное поведение и границы и убедиться в том, что они сохраняются по мере изменения кода.

Другие тесты красивы тем, что в процессе написания помогают улучшить код, для тестирования которого тонкими, но мощными способами они предназначены. Они могут не обнаруживать характерные типы ошибок и дефектов, но при этом выявлять проблемы с конструкцией, удобством тестирования или сопровождения кода; они помогают сделать ваш код более красивым.

И наконец, некоторые тесты красивы широтой своего диапазона и доскональностью. Они помогают приобрести уверенность в том, что функциональность и производительность кода отвечает всем требованиям и ожиданиям не только для горстки примеров, но и в широком диапазоне входных данных и условий.

Разработчики, желающие создавать красивый код, могут кое-чему научиться у художников, которые периодически откладывают в сторону кисти, отходят подальше от холста, обходят его кругом, вздывают свои головы, прищуриваются и смотрят на него под разными углами и под разным освещением. В поисках красоты им нужно выработать и собрать воедино все увиденное с разных точек зрения. Если вашим холстом будет интегрированная среда разработки, а средством выражения — программный код, подумайте о тестировании как о способе отойти от холста, чтобы окунуть свою работу критическим взглядом и с разных точек зрения — это сделает вас более искусным программистом и поможет создать более красивый код.

8

Динамическая генерация кода для обработки изображений

Чарльз Петцольд (*Charles Petzold*)

Среди перлов мудрости и глупости, занесенных в классическую хронологию Стивена Леви (Steven Levy) «Hackers: Heroes of the Computer Revolution» (Doubleday), мне больше всего понравилось высказывание Билла Госпера (Bill Gosper): «Данные — это всего лишь лишенная смысла разновидность программирования». Из этого, конечно же, следует, что код — это всего лишь наполненная смыслом разновидность данных, предназначенных для запуска процессоров на выполнение конкретных полезных или развлекающих действий.

Код и данные обычно абсолютно разобщены; даже в объектно-ориентированном программировании они выполняют свою собственную особую роль. Любое смешивание этих двух категорий, как это было с данными в машинных кодах, рассматривается как нарушение естественного хода вещей.

Но иногда в этом барьере между кодом и данными возникает брешь. Авторы компиляторов пишут программы, которые считывают исходный код и генерируют машинный код, но компиляторы не нарушают разделение кода и данных. Если ввод и вывод для программистов является кодом, то для компиляторов это всего лишь данные. При какой-нибудь другой дополнительной работе, осуществляющей, к примеру, дизассемблерами или симуляторами, машинный код считывается в виде данных.

Если все воспринимать рационально, без эмоций, то код и данные — это в конечном счете просто байты, а их разновидностей во всей вселенной всего лишь 256. Но смысл и значение создаются не байтами как таковыми, а тем, как они выстроены.

В отдельных случаях программы, не являющиеся компиляторами, могут извлечь из этого пользу, генерируя код в процессе своей работы. Подобная динамическая генерация кода — дело непростое, и поэтому она чаще всего находит свое применение при исключительных обстоятельствах.

На протяжении всей этой главы мы будем рассматривать пример, в котором нашел свое воплощение наиболее распространенный повод для использования

динамической генерации кода. В этом примере строго ограниченная по времени выполнения подпрограмма должна выполнять множество повторяющихся операций. В исполнении этих операций задействовано множество обобщенных параметров, и подпрограмма могла бы работать намного быстрее, если бы мы заменили эти параметры какими-то определенными значениями. Заменить параметры при написании подпрограммы не представляется возможным, поскольку до ее запуска о них ничего не известно, и от вызова к вызову они могут изменяться. Тем не менее сама подпрограмма может генерировать код в процессе своей работы. Иными словами, подпрограмма в процессе выполнения способна провести исследование своих собственных параметров, сгенерировать более эффективный код, а затем выполнить то, что получено в результате этого процесса.

Впервые я столкнулся с этой технологией при написании ассемблера. У меня была подпрограмма, осуществляющая массу повторяющихся операций. В критический момент подпрограмма должна была выполнить либо побитовую операцию AND, либо побитовую операцию OR, в зависимости от некоторого другого значения, сохраняющего постоянство во время проведения этих операций. Тестирование этого значения для выполнения AND- или OR-операции было организовано внутри цикла, что само по себе занимало много времени. Я уже подумывал о том, чтобы разбить подпрограмму на две абсолютно разные части, одну – с операцией AND, а другую – с операцией OR, но потом понял, что подпрограмма может начинаться с тестирования значения, а затем вставлять машинную команду AND или OR непосредственно в поток исполняемых команд.

Более широкое применение технологии динамической генерации кода нашла в первом выпуске Microsoft Windows (версия 1.0), который вышел в ноябре 1985 года и с тех пор начал приобретать некоторый весьма умеренный успех на рынке персональных компьютеров.

С точки зрения программиста, в первой версии Windows предлагалось примерно 200 функций для создания графических интерфейсов пользователя и для отображения векторной и растровой графики как на экране, так и на принтере в независимом до некоторой степени от конкретного устройства виде.

Среди графических функций Windows 1.0 была одна по имени BitBlt, названная в честь машинной команды оригинальной системы Xerox Alto, поддерживающей *перенос битовых блоков*. Чаще всего BitBlt использовалась для формирования растровых изображений на экране и принтере, но она также использовалась внутри Windows для отображения многих объектов пользовательского интерфейса. В общем, BitBlt переносит прямоугольные массивы пикселов от источника получателю. Связанная с ней функция по имени StretchBlt была способна во время этого процесса вытянуть или сжать исходные пиксели в больший или меньший прямоугольник места назначения.

Если используемый в BitBlt источник представлял собой растровый образ и если получателем являлся дисплей, BitBlt копировал пиксели из растрового образа на дисплей. Если источником являлся дисплей, а получателем – растровый образ, BitBlt копировал пиксели с экрана в растровый образ. Тогда растровый образ представлял собой захваченное экранное изображение.

Но если создавать подпрограмму вроде BitBlt, то можно представить себе какие-то дополнительные значения и утилиты, которые выходят за пределы простой передачи битов. Предположим, что вам нужно получить дополнительную возможность по инвертированию пикселов при их передаче от источника к получателю, чтобы черные пиксели стали белыми, светло-серые — темно-серыми, а зеленые — пурпурными.

А затем предположим, вам стало известно, что ваш коллега будет очень рад, если BitBlt сможет при передаче пикселов проверить получателя и передать пиксели от источника получателю только в том случае, если пиксели в каждой конкретной точке получателя имеют черный цвет. Это свойство позволило бы отображать непрямоугольные изображения. Например, на экране могла бы быть нарисована черная окружность, а затем BitBlt мог бы отобразить растровый образ только внутри этой окружности.

А затем кто-нибудь еще попросит создать дополнительное свойство, сочетающее в себе ранее упомянутые, благодаря чему BitBlt сможет инвертировать пиксели источника, когда получатель состоит из черных пикселов.

Приступая к изучению таких свойств, вы могли бы найти способ свести их воедино. Рассмотрим монохромную графическую систему. Каждый пиксель — это один бит, где 0 означает черный цвет, а 1 — белый. В такой системе растровый образ источника представляет собой массив однобитных пикселов, и экран тоже состоит из массива однобитных пикселов. Цвет пикселя в конкретном месте получателя зависит от значения пикселя источника (0 или 1) и значения пикселя получателя (0 или 1).

Результат в получателе для каждой конкретной комбинации пикселов источника и получателя называется растровой операцией, и их бывает, как показано в табл. 8.1, всего 16.

Таблица 8.1. Основные растровые операции

Возможные комбинации		
Входной параметр	Входное значение	
Источник (S):	1 1 0 0	
Получатель (D):	1 0 1 0	
Операция	Итог	Логическое представление
Растровая операция 0:	0 0 0 0	0
Растровая операция 1:	0 0 0 1	$\sim(S \mid D)$
Растровая операция 2:	0 0 1 0	$\sim S \wedge D$
Растровая операция 3:	0 0 1 1	$\sim S$
Растровая операция 4:	0 1 0 0	$S \wedge \sim D$
Растровая операция 5:	0 1 0 1	$\sim D$
Растровая операция 6:	0 1 1 0	$S \wedge D$
Растровая операция 7:	0 1 1 1	$\sim(S \wedge D)$
Растровая операция 8:	1 0 0 0	$S \wedge D$
Растровая операция 9:	1 0 0 1	$\sim(S \wedge D)$
Растровая операция 10:	1 0 1 0	D
Растровая операция 11:	1 0 1 1	$\sim S \mid D$
Растровая операция 12:	1 1 0 0	S

Возможные комбинации		
Операция	Итог	Логическое представление
Растровая операция 13:	1 1 0 1	S -D
Растровая операция 14:	1 1 1 0	S D
Растровая операция 15:	1 1 1 1	1

Существует четыре возможных комбинации пикселов источника и получателя, и каждая растровая операция делает из этих четырех комбинаций что-нибудь особенное, поэтому их общее количество равно 2^4 , или 16. Каждая из шестнадцати возможных растровых операций идентифицируется номером, который соотносится с шаблоном показанных в таблице итоговых пикселов. В столбце «Логическое представление» в синтаксисе языка Си показана текущая логическая операция, производимая над пикселями источника и получателя.

Например, в *растровой операции 12* (наиболее распространенной) источник просто передается получателю, а в *растровой операции 14* источник передается получателю только в том случае, если тот имеет черный цвет. *Растровая операция 10* не меняет состояние получателя, независимо от состояния источника. *Растровые операции 0* и *15* просто устанавливают цвет получателя в черный или белый соответственно и, опять же, независимо от состояния источника.

В цветной графической системе каждый пикセル, как правило, имеет глубину в 24 бита, по 8 бит на каждый основной цвет – красный, зеленый и синий. Если все биты равны нулю, цвет будет черным, а если все они равны единице, цвет будет белым. Растровые операции проводятся над соответствующими битами источника и получателя.

К примеру, при выполнении растровой операции 14 источник отображается в получателе в определенных областях, которые до этого имели черный цвет. Области получателя, которые до этого были белыми, останутся без изменений. Однако если область получателя была красной, а источник синим, то в результате получится сочетание красного и синего, то есть пурпурный цвет. Результат отличается от примера с монохромным дисплеем, но он по-прежнему полностью предсказуем.

В Windows растровые операции для BitBlt и StretchBlt усложнились еще больше. Windows поддерживает графические объекты, называющиеся *шаблоном закрашивания*, или *кистью*, которые обычно используются для заполнения замкнутых областей. Этот шаблон может иметь чистый цвет или повторяющийся рисунок, к примеру, решетку или кирпичи. Для выполнения подобных действий BitBlt и StretchBlt выполняют растровую операцию между источником, получателем и конкретным шаблоном. Этот шаблон дает возможность программе изменять пиксельные биты источника (возможно, инвертируя их или проводя преобразования по маске) безотносительно состояния получателя.

Поскольку в растровых операциях, выполняемых BitBlt и StretchBlt, задействуются три объекта – источник, получатель и шаблон, они получили название *троичных* растровых операций. Существует 256 возможных троичных растровых операций, и BitBlt и StretchBlt поддерживают каждую из них.

Как и в предыдущем случае, суть этих 256 троичных растровых операций проще понять, если начинать с рассмотрения монохромной графической системы. Так же как источник и получатель, шаблон тоже является массивом однобитных

пикселов; при визуализации шаблон накладывается на поверхность получателя. В табл. 8.2 показаны выборки из 256 возможных способов сочетаний нулевых и единичных пикселов шаблона, источника и получателя.

Таблица 8.2. Троичные растровые операции

Входной параметр	Возможные комбинации							
	Входное значение							
Шаблон (<i>P</i>):	1	1	1	1	0	0	0	0
Источник (<i>S</i>):	1	1	0	0	1	1	0	0
Операция								Итог
Растровая операция 0x00:	0	0	0	0	0	0	0	0
Растровая операция 0x01:	0	0	0	0	0	0	1	
Растровая операция 0x02:	0	0	0	0	0	1	0	
...
Растровая операция 0x60:	0	1	1	0	0	0	0	0
...
Растровая операция 0xFD:	1	1	1	1	1	0	1	
Растровая операция 0xFE:	1	1	1	1	1	1	0	
Растровая операция 0xFF:	1	1	1	1	1	1	1	1

В таблице показан пример входных данных, за которым следуют итоговые значения получателя для 7 из 256 возможных троичных растровых операций. Каждая из этих операций может быть идентифицирована по однобайтовому шестнадцатеричному номеру, соотносящемуся с шаблоном итоговых битов получателя, который показан в таблице. Например, для *растровой операции 0x60*, если пиксель шаблона имеет значение 1 (белый), пиксель источника имеет значение 0 (черный), а пиксель получателя имеет значение 1, то итоговый пиксель получателя будет иметь значение 1 (белый).

В ранних версиях Windows 15 из 256 растровых операций идентифицировались по именам, как в документации, так и в файле заголовка Windows, который использовался программистами, работающими на языке Си. Первая из них, где получатель в итоге раскрашивается всеми нулями независимо от состояний шаблона, источника и получателя, была известна как BLACKNESS, а последняя – как WHITENESS.

Справочник программирования под Windows идентифицирует все 256 растровых операций по побитовым логическим операциям, которые ими выполняются, выраженным в обратной польской нотации. Например, растровая операция 0x60 соответствует логической операции PDSxa.

Это означает, что операция Исключающее ИЛИ (*x*) выполняется между получателем (*D*) и источником (*S*), а результат смешивается с шаблоном (*P*) в побитовой операции И (*a*). В цветных системах такие же логические операции выполняются над цветовыми битами источника, получателя и шаблона. В таком же написании эти растровые операции задокументированы на веб-сайте по адресу: http://msdn.microsoft.com/library/en-us/gdi/pantdraw_6n77.asp.

Некоторые из этих растровых операций при определенных обстоятельствах могут оказаться весьма полезными. Например, вам может потребоваться инвер-

тировать те пиксели получателя, которые соответствуют областям, где кисть рисует черным цветом, но показывать растрочный образ источника в тех областях, где кисть рисует белым цветом. Это соответствует растроевой операции 0xC5. Разумеется, многие из 256 возможных операций не находят широкого практического применения, я даже подозреваю, что большинство из них нигде, кроме демонстрационного и исследовательского кода, никогда не использовались. Тем не менее было полностью удовлетворено чувство законченности и универсальности.

А если бы мы сами взялись за реализацию универсальной функции BitBlt, то как бы мы это сделали?

Предположим, что сейчас 1985 год и мы используем язык программирования Си. Давайте также в демонстрационных целях предположим, что работа ведется с графической системой, построенной на градациях серого цвета и имеющей один байт на пикセル, и что источник, получатель и шаблон могут быть доступны через двумерные массивы, названные *S*, *D* и *P*. То есть каждая из этих переменных является указателем на семейство байтовых указателей, каждый из которых указывает на начало горизонтальной строки пикселов, таким образом *S[y][x]* предоставляет доступ к байту в строке *y* и столбце *x*. Ширина и высота рабочей области хранится в *cx* и *cy* (это соответствует сложившемуся при программировании в среде Windows соглашению об именах переменных: *c* означает счетчик, следовательно, эти переменные являются признаком счетчика значений *x* и *y*, или ширины и высоты). В переменной *gor* хранится код раstroвой операции, принимающий значения от 0 до 255.

Следующий простой код на языке Си предназначен для осуществления пересылки, подобной той, что используется в BitBlt. В операторе *switch* для определения, какую операцию нужно выполнять для вычисления значения пикселя получателя, используется переменная *gor*. Здесь показаны только 3 из 256 раstroвых операций, позволяющие понять замысел в общих чертах:

```
for (y = 0; y < cy; y++)
for (x = 0; x < cx; x++)
{
    switch(rop)
    {
        case 0x00:
            D[y][x] = 0x00;
            break;
        ...
        case 0x60:
            D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];
            break;
        ...
        case 0xFF:
            D[y][x] = 0xFF;
            break;
    }
}
```

Код, конечно, *симпатичный*, то есть он хорошо выглядит, его предназначение и функциональность не вызывают никаких сомнений. Но красивым его *не назовешь*, поскольку красивый код должен вас устраивать своей работой.

А этот код – настоящая *катастрофа*, поскольку он работает с растровыми изображениями, объем которых может быть *очень большим*. В наши дни растровые изображения, которые создаются недорогими цифровыми камерами, могут иметь *миллионы* пикселов. Неужели вам действительно захочется, чтобы оператор `switch` оказался внутри циклов перебора строк и столбцов?

И должна ли вообще логическая структура `switch` выполнятся для каждого отдельного пикселя? Наверное нет. Конечно, перемещение циклов внутрь каждого оператора `case` загромождает код, но теперь, по крайней мере, у него есть шансы побороться за приемлемую производительность:

```
switch(rop)
{
case 0x00:
    for (y = 0; y < cy; y++)
        for (x = 0; x < cx; x++)
            D[y][x] = 0x00;
    break;
...
case 0x60:
    for (y = 0; y < cy; y++)
        for (x = 0; x < cx; x++)
            D[y][x] = (D[y][x] ^ S[y][x]) & P[y][x];
    break;
...
case 0xFF:
    for (y = 0; y < cy; y++)
        for (x = 0; x < cx; x++)
            D[y][x] = 0xFF;
    break;
}
```

Будь это в 1985 году при создании Windows, то вам не пришлось бы даже использовать язык Си. Ранние *приложения* Windows большей частью были написаны на Си, а вот сама система Windows была написана на ассемблере 8086.

Для таких важных для Windows функций, как `BitBlt`, требовалось даже более радикальное решение – нечто более быстрое, чем ассемблер, что невозможно себе даже представить. Программисты *Microsoft*, создавшие `BitBlt`, очень гордились своим творением, и те из нас, кто изучал программирование в среде Windows в середине 80-х годов прошлого века, находились под большим впечатлением, когда они хвастались своими достижениями.

Функция `BitBlt` содержала своего рода небольшой компилятор. На основе растровой операции (а также графического формата, количества бит на пикセル и размеров графической области) функция `BitBlt` транслировала машинный код 8086 в стек в виде подпрограммы, а затем выполняла эту подпрограмму. Эта палиативная подпрограмма в машинных кодах перебирала в цикле все пиксели и осуществляла запрошенную растровую операцию.

Для осуществления `BitBlt` и ее 256 растровых операций это было отличное решение. Хотя этот мини-компилятор требовал небольших дополнительных затрат на помещение машинного кода в стек, попиксельная обработка достигала максимально возможной скорости, а это при работе с растровыми образами – самое важное свойство. Более того, код `BitBlt`, имевшийся в Windows, наверняка

был значительно короче, чем должен был быть при явной реализации всех 256 растровых операций.

Изучая документацию по троичным растровым операциям, можно было получить некоторое представление о работе использовавшегося в BitBlt мини-компилятора. К примеру, растровая операция, идентифицируемая по номеру 0x60, осуществляла логическую операцию PDSxa. При вызове BitBlt на самом деле выдавался 32-битный код растровой операции, документированной как 0x00600365. Обратите внимание, что в этот номер вставлен байт 0x60, а также заметьте, что последние два байта составляют число 0x0365.

В растровой операции с результатом 11110110, или 0xF6, проводится логическая операция PDSxo, которая очень похожа на PDSxa, за исключением того, что в ней вместо операции И проводится операция ИЛИ.

Полный 32-битный код растровой операции, передающийся функции BitBlt -0x00F70265. Последние два байта составляют число 0x0265, которое очень похоже на число 0x0365 операции PDSxa. Если исследовать большее число 32-битных кодов растровых операций, станет абсолютно ясно, что сам код растровой операции служит своеобразным шаблоном для мини-компилятора BitBlt, служащим для сборки нужного машинного кода. Эта технология экономит для BitBlt как память, так и время, необходимое для использования поисковой таблицы.

Конечно, Windows 1.0 была создана свыше 20 лет назад. С тех пор все, включая Windows, ушло далеко вперед. Сегодня в языках программирования я отдаю предпочтение C#, а не ассемблеру или Си. Обычно я пишу то, что называется управляемым кодом, который запускается под управлением Microsoft .NET Framework. Компилятор C# превращает мой исходный код в не зависящий от используемого процессора промежуточный язык – Intermediate Language (часто называемый Microsoft Intermediate Language, или MSIL). И только потом, после запуска программы, .NET Common Language Runtime использует компилятор времени выполнения для преобразования Intermediate Language в машинный код, соответствующий тому процессору, на котором выполняется программа.

И все же все виды цифровой обработки изображений все еще не могут обойтись без нестандартных подходов к программированию. При работе с миллионами пикселов, попиксельная обработка должна осуществляться все быстрее, быстрее и быстрее. При разработке коммерческого продукта, вы, скорее всего, захотите нанять программиста, работающего на ассемблере, или кого-нибудь, кто знает, как нацелить на обработку графический процессор (Graphics Processing Unit, GPU), находящийся на видеокартах. Даже для разрабатываемого по случаю или некоммерческого программного продукта вам наверняка захочется воспользоваться чем-нибудь более быстродействующим, чем обычная циклическая обработка, заданная на языке высокого уровня.

Недавно, экспериментируя с кодом на C#, реализующим цифровые фильтры изображений, называемые также фильтрами изображений, или цифровыми фильтрами, я вспомнил о динамической генерации кода, использованной в исходной Windows-функции BitBlt. Растровые операции, подобные тем, что были реализованы в Windows-функциях BitBlt и StretchBlt, используют только соответствующие

пиксели источника, получателя и шаблона. Цифровые фильтры затрагивают ближайшие пиксели. Конкретный цифровой фильтр применяется к растровому изображению, чтобы внести в него какие-нибудь изменения, возможно, повысить четкость по краям или сделать расплывчатым все изображение. К примеру, фильтр расплывчатости усредняет состояния ближайших пикселов, чтобы вычислить значение пикселя-получателя.

Простые цифровые фильтры изображений зачастую реализуются в виде небольших массивов чисел. Эти массивы обычно квадратные по структуре и имеют нечетное количество строк и столбцов. На рис. 8.1 показан простой пример такого массива.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Рис. 8.1. Простое цифровое изображение

Изображение на рис. 8.1 представляет собой фильтр 3×3 , и его можно рассматривать как средство преобразования исходного растрового изображения в целевое. Совместите центр фильтра с каждым пикселиом исходного изображения, чтобы остальные восемь ячеек были совмещены с окружающими пикселями. Перемножьте девять значений фильтра с девятью пикселями источника и сложите результаты. Это будет соответствовать пикселиу целевого изображения. Если в пикселях закодирован цвет или прозрачность, то фильтр нужно применить к каждому цветовому каналу по отдельности. У некоторых фильтров имеются разные массивы для разных цветовых каналов или их реализация по определенному алгоритму, но в своих упражнениях мы остановимся на самых простых фильтрах.

Фильтр, во всех ячейках которого содержится значение $1/9$, применяется для придания изображению расплывчатости. Каждый пиксель целевого изображения является усредненным результатом соседних девяти пикселов исходного изображения. Хорошо, что все числа в сумме составляют единицу, в результате этого изображение не становится ни светлее, ни темнее, но этот фильтр вполне может содержать все единицы или другие числа. Все, что потребуется для компенсации, — это поделить итоговую сумму на количество ячеек фильтра (скоро станет понятно, что я предпочитаю именно этот метод).

На рис. 8.2 показан фильтр, повышающий резкость изображения. Этот фильтр стремится выделить высококонтрастные области.

Предположим, что мы работаем с изображениями, выполненными в градациях серого цвета, с одним байтом на пиксель. Пиксели исходного изображения сохранены в двумерном массиве под названием S . Наша задача заключается

в вычислении значений для пикселов целевого массива под названием D. Горизонтальные и вертикальные размеры обоих массивов сохранены в переменных cxBitmap и cyBitmap. Есть также двумерный массив Filter под названием F, с размерами, указанными в cxFilter и cyFilter. В примере 8.1 показан простой пример кода на языке Си, предназначенный для применения фильтра.

0	-1	0
-1	4	-1
0	-1	0

Рис. 8.2. Фильтр резкости

Пример 8.1. Простейший Си-код для применения цифрового фильтра

```

for (yDestination = 0; yDestination < cyBitmap; yDestination++)
for (xDestination = 0; xDestination < cxBitmap; xDestination++)
{
    double pixelsAccum = 0;
    double filterAccum = 0;

    for (yFilter = 0; yFilter < cyFilter; yFilter++)
    for (xFilter = 0; xFilter < cxFilter; xFilter++)
    {
        int ySource = yDestination + yFilter - cyFilter / 2;
        int xSource = xDestination + xFilter - cxFilter / 2;
        if (ySource >= 0 && ySource < cyBitmap &&
            xSource >= 0 && xSource < cxBitmap)
        {
            pixelsAccum += F[y][x] * S[y][x];
            filterAccum += F[y][x];
        }
    }
    if (filterAccum != 0)
        pixelsAccum /= filterAccum;

    if (pixelsAccum < 0)
        D[y][x] = 0;
    else if (pixelsAccum > 255)
        D[y][x] = 255;
    else
        D[y][x] = (unsigned char) pixelsAccum;
}

```

Обратите внимание, что циклический перебор фильтра приводит к накоплению двух итоговых сумм. В переменной pixelsAccum накапливается сумма, полученная на основе значений пикселов исходного изображения и ячеек фильтра, а в переменной filterAccum накапливается сумма, полученная на основе только

ячеек фильтра. Для целевых пикселов, находящихся по краям изображения, некоторые ячейки фильтра соотносятся с пикселями за пределами пространства исходного изображения. Я предпочитаю игнорировать эти ячейки, ничего не добавляя к `pixelsAccum` и `filterAccum`, а потом разделить `pixelsAccum` на `filterAccum`, чтобы обеспечивать примерную корректность целевого пикселя. Именно поэтому `filterAccum` не вычисляется вне цикла, а ячейки фильтра не должны быть нормализованы дополнением до единицы. Также обратите внимание, что ближе к концу кода отношение `pixelsAccum` к `filterAccum` должно быть зафиксировано между 0 и 255, чтобы не вызвать каких-нибудь необычных эффектов.

Для каждого пикселя целевого изображения требуется девять раз обратиться и к исходному изображению, и к фильтру. Более того, с ростом разрешения растрового изображения фильтры зачастую также должны укрупняться, чтобы производить на изображение более заметное воздействие.

Для языка высокого уровня это довольно большой объем обработки, но мне было интересно посмотреть, как C# и .NET справляются с трудностями. Эксперименты с обработкой изображений на C# я начал с кода Windows Forms из моей книги «Programming Windows with C#» (Microsoft Press). Программа `ImageClip` из главы 24 этой книги объединяет код для загрузки, просмотра, распечатки и сохранения растровых изображений различных популярных форматов, включая JPEG, GIF и PNG. Этот код наряду с тем кодом, который я написал для нашего упражнения, пригоден для загрузки и использования в программе под названием `ImageFilterTest`. Для компиляции файла проекта требуется Visual Studio 2005, а исполняемый файл должен запускаться под .NET Framework 2.0 или более поздней версии. Для использования программы нужно выполнить следующие действия.

1. В меню `File` (Файл) выбрать пункт `Open` (Открыть) и загрузить в нее полноцветное растровое изображение. Имеющийся в программе код фильтров работает только с изображениями глубиной 24 или 32 бита на пикセル; он не работает с изображениями, использующими таблицы цветовых палитр, включая те, в которых таблица палитры содержит оттенки серого.
2. Выберите один из фильтров в меню `Filter` (Фильтр). Фильтр будет применен к изображению, и программа сообщит о том, сколько времени на это было затрачено. Первый пункт меню `Filter` (Фильтр), `Use method that generates Intermediate Language` (Использовать метод, генерирующий Intermediate Language), позволяет выбрать метод, применяющий фильтр. По умолчанию программа использует метод под названием `FilterMethodCS` (сокращение, означающее «метод фильтра, использующий C#»). Если будет задействован первый пункт меню, программа воспользуется методом `FilterMethodIL` («метод фильтра, использующий Intermediate Language»). Чуть позже в этой главе будут описаны оба этих метода.

Если вы пытаетесь написать на C# высокопроизводительный код, то одним из наиболее интересных занятий будет анализ скомпилированного файла с использованием небольшой утилиты под названием *IL Disassembler*, включенной в пакет разработчика – .NET Software Development Kit. Этот дизассемблер показывает код *Intermediate Language*, сгенерированный компилятором C#. Хотя

программа не показывает финальной стадии — преобразование кода Intermediate Language в машинный код компилятором времени исполнения — вы можете ею воспользоваться для определения некоторых проблемных областей. В самом начале я разочаровался в идее хранения пикселов изображения в двумерных массивах. C# поддерживает многомерные массивы, но на уровне Intermediate Language помещение элементов в многомерные массивы и их извлечение требуют вызова определенных методов. Тем не менее команды Intermediate Language поддерживают доступ к одномерным массивам. К тому же в стандартном (и быстродействующем) коде пересылки пикселов из объекта растрового изображения в массив и обратно задействуется одномерный массив. Выполнение кода, который я написал, чтобы передать все в двумерный массив, само по себе занимало значительное количество времени.

Чтобы инкапсулировать фильтры изображений и методы, применяющие эти фильтры к изображениям, я создал класс под названием `ImageFilter`, содержащий три частных поля и конструктор, который присваивает этим полям значения. Частное поле `filter` представляет собой одномерный массив, содержащий двумерный фильтр, поэтому для обозначения подразумеваемых номеров столбцов и строк возникла необходимость в полях `cxFilter` и `cyFilter`:

```
class ImageFilter
{
    double[] filter;
    int cxFilter;
    int cyFilter;

    public ImageFilter(int cxFilter, double[] filter)
    {
        this.filter = filter;
        this.cxFilter = cxFilter;
        this.cyFilter = filter.Length / cxFilter;
    }

    ...
}
```

Если разрешены будут только квадратные фильтры, то параметр `cxFilter` конструктору не понадобится, а число строк и столбцов может быть без труда вычислено извлечением квадратного корня из размера массива `filter`, возвращаемого методом `filter.Length`. Параметр `cxFilter` допускает использование для фильтров не только квадратных, но и прямоугольных массивов. Если `cxFilter` показывает количество столбцов фильтра, количество строк вычисляется в выражении `filter.Length/cxFilter`, которое в моем коде заведомо считается целым числом.

Класс `Filter` содержит метод под названием `ApplyFilter`, имеющий параметр типа `Bitmap`. Я не хочу здесь показывать метод `ApplyFilter`, поскольку в нем просто содержится стандартный код, который сначала получает доступ к пикселям объекта `Bitmap` (используя метод под названием `LockBits`), а затем преобразует пиксели в одномерный массив. Второй параметр метода `ApplyFilter` имеет тип `Boolean` и называется `willGenerateCode`. Если он имеет значение `false`, метод `ApplyFilter` вызывает `FilterMethodCS`.

Метод `FilterMethodCS`, показанный в примере 8.2, по сути является реализацией алгоритма применения фильтра из примера 8.1, но переведенного на язык C# и использующего одномерные массивы.

Пример 8.2. Алгоритм цифрового фильтра, реализованный на C#

```

1 void FilterMethodCS(byte[] src, byte[] dst, int stride, int bytesPerPixel)
2 {
3     int cBytes = src.Length;
4     int cFilter = filter.Length;
5
6     for (int iDst = 0; iDst < cBytes; iDst++)
7     {
8         double pixelsAccum = 0;
9         double filterAccum = 0;
10
11        for (int iFilter = 0; iFilter < cFilter; iFilter++)
12        {
13            int yFilter = iFilter / cyFilter;
14            int xFilter = iFilter % cxFilter;
15
16            int iSrc = iDst + stride * (yFilter - cyFilter / 2) +
17                         bytesPerPixel * (xFilter - cxFilter / 2);
18
19            if (iSrc >= 0 && iSrc < cBytes)
20            {
21                pixelsAccum += filter[iFilter] * src[iSrc];
22                filterAccum += filter[iFilter];
23            }
24        }
25        if (filterAccum != 0)
26            pixelsAccum /= filterAccum;
27
28        dst[iDst] = pixelsAccum < 0 ? (byte)0 : (pixelsAccum > 255 ?
29                                         (byte)255 : (byte)pixelsAccum);
30    }
31 }
```

Первые два параметра — это исходный и целевой массивы `src` и `dst`. Третий параметр — `stride`, представляет собой количество байтов в каждой строке исходного и целевого изображения. Значение `stride` обычно равно ширине изображения в пикселях, помноженной на количество байтов на пиксел, и если бы не соображения производительности, оно могло бы быть округлено в большую сторону, до предела в четыре байта. (Поскольку программа работает только с полноцветными изображениями, количество байтов на пиксел будет всегда равняться трем или четырем.) Вычислять значение `stride` нет необходимости, поскольку в этот параметр попадают сведения, возвращенные методом `LockBits` при получении доступа к битам изображения. Чтобы избежать частого обращения к свойству `Length`, метод начинается с сохранения количества байтов, имеющегося в массивах `src` и `filter`. Переменные, чье имя начинается на букву `i`, являются индексами для трех массивов, задействованных в этом методе.

Если преследуемая цель заключается в написании быстродействующего алгоритма цифрового фильтра, то метод `FilterMethodCS` ее выполнение не обеспе-

чит. При изображении в 300 000 пикселов с глубиной в 24 бита на пиксел и фильтром 5×5 для работы этого метода на моем Pentium 4 с тактовой частотой 1,5 ГГц потребуется около двух секунд. Возможно, две секунды — это не так уж и плохо, но фильтр 5×5 , примененный к 4,3-мегапиксельному изображению с глубиной в 32 бита на пиксел, потребует около полминуты, а это уже очень много. И я не вижу способа улучшения кода C#, позволяющего увеличить его эффективность.

По традиции, если функция работает недостаточно быстро и вы чувствуете, что не в состоянии провести ее дальнейшую оптимизацию, то рассматривается возможность применения языка ассемблера. В эру независимости от применяемой платформы и управляемого кода вместо этого можно рассмотреть в качестве своеобразного эквивалентного подхода написание подпрограммы непосредственно на языке .NET Intermediate Language. Вне всякого сомнения это вполне состоятельное решение, и оно может даже рассматриваться чем-то вроде развлечения (при соответствующем уровне развития).

Тем не менее даже программирования на Intermediate Language может быть недостаточно. Воспользуйтесь дизассемблером IL Disassembler и посмотрите на код Intermediate Language, сгенерированный компилятором C# для метода FilterMethodCS. Вы действительно думаете, что сможете его существенно улучшить?

Проблема метода FilterMethodCS состоит в том, что он рассчитан на обработку изображений и применение фильтров любых размеров. Основная часть кода FilterMethodCS занята «напряженной работой» с циклами и индексами. Метод можно существенно улучшить, если лишить его этой универсальности. Предположим, что вы всегда работаете с изображениями глубиной 32 бита на пиксел одинакового размера, который я обозначу как CX и CY (считайте эти прописные буквы чем-то вроде #defines в Си или C++ или значений const в C#). И предположим, что вы всегда используете один и тот же фильтр размером 3×3 с фиксированными элементами, чьи поля обозначены, как на рис. 8.3.

F11	F12	F13
F21	F22	F23
F31	F32	F33

Рис. 8.3. Формат массива для фильтра 3×3

Как теперь вы подойдете к написанию метода применения фильтра? Можно обойтись без цикла iFilter и жестко задать логику для девяти элементов фильтра:

```
// Ячейка фильтра F11
int iSrc = iDst - 4 * CX - 4;

if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F11;
    filterAccum += F11;
}
```

```

// Ячейка фильтра F12
iSrc = iDst - 4 * CX;

if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F12;
    filterAccum += F12;
}

// Ячейки фильтра с F13 по F32
...
// Ячейка фильтра F33
iSrc = iDst + 4 * CX + 4;

if (iSrc >= 0 && iSrc < 4 * CX * CY)
{
    pixelsAccum += src[iSrc] * F33;
    filterAccum += F33;
}

```

При таком подходе мы избавляемся от логики цикла, упрощаем вычисление *iSrc* и исключаем и устранием доступ к массиву *filter*. Хотя код становится явно большим по объему, его быстродействие, несомненно, увеличивается. Поскольку вам известны значения всех элементов фильтра, можно несколько сократить код, исключив те случаи, когда элементы фильтра содержат нуль, или упростить те случаи, когда они содержат 1 или -1.

Разумеется, задание жесткой логики — шаг непрактичный, поскольку в действительности нужно иметь возможность работы с множеством различных по размеру изображений и множеством типов фильтров, свойства которых неизвестны до тех пор, пока дело не дойдет до их применения.

Вместо применения жестко заданной логики фильтра гораздо лучшим подходом может стать генерация специализированного кода на лету, основываясь на размере и пиксельной глубине изображения, а также на размере и содержимом элементов фильтра. В былые времена вы могли бы все сделать так же, как разработчики Windows сделали с *BitBlt*, то есть сгенерировать машинный код в памяти и запустить его на выполнение. Если перевести все это на современные рельсы и позаботиться о переносимости кода, решение может заключаться в генерации в C# кода .NET Intermediate Language с его последующим выполнением.

Это вполне осуществимое решение. В программе C# вы можете создать в памяти статический метод, который состоит из команд Intermediate Language, а затем этот метод выполнить, в этот самый момент имеющийся в .NET JIT-компилятор вводит изображение для преобразования кода Intermediate Language в машинный код. Но из этого не следует, что в ходе всего этого процесса вы должны отказаться от написания управляемого кода.

Возможность динамической генерации кода на Intermediate Language была представлена в .NET 2.0, и в ней задействуются классы из пространства имен *System.Reflection.Emit*. Вы можете генерировать целые классы и даже полные

сборки, но для небольших приложений (к которым относится и наша разработка) вы можете просто сгенерировать статический метод, а затем вызвать его на выполнение. Именно это я и сделал в методе `FilterMethodIL` класса `ImageFilter`.

Здесь я собираюсь показать вам весь метод `FilterMethodIL` (исключив множество комментариев, имеющихся в исходном программном файле `ImageFilter.cs`), поскольку в нем задействуются некоторые моменты весьма интересного взаимодействия между кодом C# и сгенерированным кодом на Intermediate Language. Изучая метод `FilterMethodIL`, нужно постоянно помнить о том, что код Intermediate Language генерируется в нем при каждом применении определенного фильтра к определенному изображению, поэтому в этом коде все показатели фильтра, а также размер и пиксельная глубина изображения жестко заданы.

Очевидно, для генерации этого кода потребуются некоторые непроизводительные издержки, но они ничтожны по сравнению с тем количеством операций, которое требуется для больших изображений, которые запросто могут содержать более чем миллион пикселов.

Весь код `FilterMethodIL` показан ниже последовательными частями, сопровождаемыми пояснениями, описывающими все в них происходящее и представляющими некоторые новые понятия. Метод `FilterMethodIL` имеет те же параметры, что и метод `FilterMethodCS`, и начинается с получения всего размера изображения в байтах:

```
void FilterMethodIL(byte[] src, byte[] dst, int stride, int bytesPerPixel)
{
    int cBytes = src.Length;
```

Чтобы создать в коде статический метод, нужно создать объект типа `DynamicMethod`. Во втором аргументе конструктора указывается тип данных, возвращаемых методом, а третий аргумент представляет собой массив из типов параметров метода. Четвертый аргумент — это класс, создающий этот метод, и он доступен из метода `GetType`:

```
DynamicMethod dynameth = new DynamicMethod("Go", typeof(void),
    new Type[] { typeof(byte[]), typeof(byte[]) }.GetType());
```

Как видно из третьего аргумента конструктора, оба параметра в этом динамическом методе представляют собой байтовые массивы. Это будут массивы `src` и `dst` из примера 8.2. В коде Intermediate Language эти два параметра будут упомянуты под индексами 0 и 1.

Для генерации кода Intermediate Language, содержащего тело этого метода, вы получаете объект типа `ILGenerator`:

```
ILGenerator generator = dynameth.GetILGenerator();
```

Впоследствии этот объект будет многократно использован. Теперь можно приступить к определению локальных переменных метода. Я решил, что будет достаточно иметь три локальные переменные, соответствующие трем локальным переменным метода `FilterMethodCS`:

```
generator.DeclareLocal(typeof(int)); // Index 0 - iDst
generator.DeclareLocal(typeof(double)); // Index 1 - pixelsAccum
generator.DeclareLocal(typeof(double)); // Index 2 - filterAccum
```

Судя по комментариям, ссылки на эти локальные переменные будут производиться по индексам. Теперь мы готовы приступить к определению цикла на

основе iDst, в котором будет осуществляться доступ ко всем пикселям целевого массива. Эти три оператора соответствуют объявлениям таких же переменных в строках 3, 4 и 6 примера 8.2.

Большая часть того, что осталось осуществить, требует генерации кодов операций Intermediate Language, которые похожи на коды операций машинного языка. Intermediate Language состоит из однобайтных кодов операций, которые иногда сопровождаются аргументами. Разумеется, вам не придется прикладывать руки ко всем этим битам и байтам. Для генерации этих кодов операций вызывается одна из перезагрузок метода `Emit`, определенного в классе `I1Generator`. Первым аргументом метода `Emit` всегда служит объект типа `OpCode`, а все доступные коды операций заранее определены как статические, доступные только для чтения поля класса `OpCodes` (обратите внимание, что название дано во множественном числе). На момент написания этих строк класс `OpCodes` был задокументирован на сетевом ресурсе <http://msdn2.microsoft.com/library/system.reflection.emit.opcodes.aspx>.

В Intermediate Language большинство присваиваний и операционной логики основано на применении виртуального стека вычислений. (Я называю его виртуальным, поскольку реальным кодом, который в конечном итоге будет выполнен процессором вашего компьютера, является машинный код, сгенерированный JIT-компилятором, и этот код может имитировать или не имитировать стек вычислений Intermediate Language.) Команда `load` проталкивает значение в стек. Это может быть либо определенное число, либо значение локальной переменной, либо что-нибудь еще. А команда `store` извлекает значение из стека и сохраняет его в локальной переменной или где-нибудь еще. Арифметические и логические операции также выполняются в стеке. К примеру, `add` выталкивает два значения из стека, складывает их и проталкивает результат в стек.

В Intermediate Language для установки значения локальной переменной `iDst` в 0 нужно воспользоваться командами `load` и `store`. Команда `Ldc_14_0` помещает значение четырехбайтового целого числа 0 в стек, а команда `Stloc_0` сохраняет это значение в локальной переменной, имеющей индекс 0, которая соответствует `iDst`:

```
generator.Emit(OpCodes.Ldc_14_0);
generator.Emit(OpCodes.Stloc_0);
```

Хотя во многих высокоуровневых языках программирования имеется команда `goto` (или ее эквивалент), современные программисты пользоваться ею не советуют. Тем не менее в ассемблере и в Intermediate Language команда `goto`, общезвестная как команда `jmp` или `branch`, – единственная доступная форма управления потоком данных. Все операторы `for` и `if` должны быть имитированы за счет переходов.

В .NET Intermediate Language поддерживается оператор безусловного перехода и несколько условных переходов. Эти условные переходы зависят от результата определенного предшествующего им сравнения. Например, команда `branch if less than` осуществляет переход, если в предыдущем сравнении одно значение было меньше, чем другое. Имитация в Intermediate Language конструкций `if` и `else` требует двух меток, одна из которых соответствует началу блока `else`, а другая указывает на позицию после блока `else`. Если условие `if` явля-

ется ложным, условный переход осуществляется на первую метку; в противном случае исполняется блок `if`. В конце блока `if` осуществляется безусловный переход на блок `else`. На рис. 8.4 показаны два возможных маршрута.

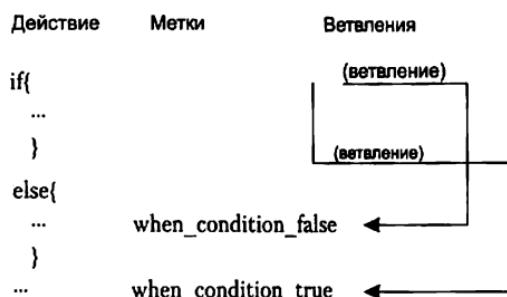


Рис. 8.4. Переходы в Intermediate Language, осуществляющие `if/else`

Реальный код команды перехода в Intermediate Language содержит числовое значение, показывающее адрес пред назначенной команды в виде смещения от адреса текущей команды. Вычисление этих смещений было бы для программистов слишком неприятным занятием, поэтому для облегчения этой задачи предусмотрена система установки меток. Все, что нужно, — это определить, где метки должны быть вставлены в поток команд таким образом, чтобы при указании перехода на эту метку генератор кода смог вычислить правильное числовое значение смещения.

Для того чтобы воспользоваться меткой, требуется вызов двух методов. Вызов метода `DefineLabel` определяет метку, на которую затем можно будет ссылаться в командах перехода. Вызов метода `MarkLabel` осуществляет вставку метки в поток команд Intermediate Language. Этот двухэтапный процесс позволяет определить метку, а затем выделить оперативный код, который осуществляет переход на эту метку, даже если бы позже метка не появилась бы в потоке команд. В следующих строках, для того чтобы поместить объект `Label`, названный `labelTop`, на вершину цикла `iDst`, вызываются оба метода — `DefineLabel` и `MarkLabel`:

```
Label labelTop = generator.DefineLabel();
generator.MarkLabel(labelTop);
```

Эта метка является эквивалентом размещения оператора `for` в строке 6 кода C# в примере 8.2. Необходимость в метке здесь вызвана тем, что код в конце цикла должен перейти на его вершину.

Теперь мы генерируем код внутри цикла `iDst`. В нем осуществляется по-пиксельная обработка. Первым делом переменным `pixelsAccum` и `filterAccum` присваивается начальное значение 0. Первый вызов `Emit`, показанный в следующем кодовом фрагменте, содержит операционный код `Ldc_R8`, который загрузит 8-битное число типа `real` (то есть число с плавающей запятой) в стек. Вторым аргументом метода `Emit` служит само это число. Тип этого числа должен соответствовать тому типу, который подразумевается в коде операции. Если воспользоваться нулем без десятичного знака, компилятор C# посчитает

его целым числом, и вы не будете знать о своем просчете до тех пор, пока вызов исключения во время выполнения не укажет на ошибку в программе:

```
generator.Emit(OpCodes.Ldc_R8, 0.0);
generator.Emit(OpCodes.Dup);
generator.Emit(OpCodes.Stloc_1);
generator.Emit(OpCodes.Stloc_2);
```

Команда Dup продублирует в стеке значение 0, а коды операций Stloc_1 и Stloc_2 сохранят значение в локальных переменных, представляющих pixelsAccum и filterAccum. Здесь вы также должны убедиться в соответствии типов, иначе во время выполнения будет вызвано исключение, указывающее, что JIT-компилятор обнаружил программу с ошибкой. Теперь мы готовы сгенерировать код для каждого элемента в массиве filter. Но заставлять Intermediate Language перебирать в цикле массив filter и получать доступ к каждому из его элементов не хочется. Вместо этого нам нужно, чтобы все элементы фильтра были в нем жестко заданы. Если фильтр имеет девять элементов, требуется девять одинаковых фрагментов кода Intermediate Language. Для этих целей мы воспользуемся оператором if языка C# и осуществим циклический перебор элементов фильтра:

```
for (int iFilter = 0; iFilter < filter.Length; iFilter++)
{
```

Если отдельный элемент фильтра будет содержать 0, его можно просто проигнорировать — генерировать код Intermediate Language не нужно, поэтому можно перейти к следующему элементу массива filter:

```
if (filter[iFilter] == 0)
    continue;
```

Для каждого элемента фильтра индекс массива src будет точным смещением от индекса iDst. Это смещение вычисляется показанным далее кодом C#. Значение offset может быть вычислено в коде C#, поскольку в коде Intermediate Language оно должно фигурировать только в виде константы:

```
int xFilter = iFilter % cxFilter;
int yFilter = iFilter / cxFilter;
int offset = stride * (yFilter - cyFilter / 2) +
    bytesPerPixel * (xFilter - cxFilter / 2);
```

Доступ к элементу массива или установка его значения происходят в три этапа. Сначала нужно поместить ссылку на массив в стек. Затем нужно поместить туда же индекс массива. И наконец, если вы получаете доступ к данному элементу, требуется команда load, а если устанавливаете его значение, то требуется команда store. Массив src должен быть доступен для каждого ненулевого элемента массива filter, поэтому теперь пришло время поместить ссылку на этот массив в стек вычислений:

```
generator.Emit(OpCodes.Ldarg_0);
```

Команда Ldarg ссылается на аргументы генерируемого метода, а массив src будет первым аргументом, который связан с индексом 0.

Затем нужно определить ряд меток. Обратите внимание, что эти три метки определены так, чтобы команды Intermediate Language могли на них ссылаться,

но они еще не расставлены, потому что будут вставлены в поток команд Intermediate Language чуть позже:

```
Label labelLessThanZero = generator.DefineLabel();
Label labelGreaterThanOrEqual = generator.DefineLabel();
Label labelLoopBottom = generator.DefineLabel();
```

Для каждого элемента фильтра массив `src` должен быть доступен через индекс `iDst` плюс значение смещения, которое уже было вычислено кодом C#.

В следующем фрагменте кода `iDst` помещается в стек, затем следует текущее значение смещения — `offset`, затем они суммируются (в результате чего два операнда выталкиваются из стека, а их сумма проталкивается в стек), и сумма дважды дублируется:

```
generator.Emit(OpCodes.Ldloc_0);           // загрузка индекса dst в стек
generator.Emit(OpCodes.Ldc_I4_Offset);       // загрузка в стек смещения
generator.Emit(OpCodes.Add);                // Суммирование двух первых значений
generator.Emit(OpCodes.Dup);                // Двойное дублирование
generator.Emit(OpCodes.Dup);
```

Результирующий индекс (который в `FilterMethodCS` назывался `iSrc`) может выходить за границы массива. В следующем фрагменте кода в стек загружается целое число, равное нулю, и осуществляется переход, если `iSrc` меньше, чем 0, при этом оба операнда выталкиваются из стека. Он служит частичным эквивалентом условий оператора `if` в строке 19 примера 8.2:

```
generator.Emit(OpCodes.Ldc_I4_0);
generator.Emit(OpCodes.Blt_S, labelLessThanZero);
```

`Blt` означает переход, если меньше чем (`branch if less than`), а `S` указывает на короткий переход (один из переходов, в котором цель находится на удалении менее, чем 256 кодов операций).

Вторая проверка определяет, не выходит ли значение `iSrc` за предел размера изображения в байтах. Обратите внимание, что для проведения сравнения в стек проталкивается само значение `cBytes`. Это оставшаяся часть условия `if` из строки 19 примера 8.2:

```
generator.Emit(OpCodes.Ldc_I4_cBytes);
generator.Emit(OpCodes.Bge_S, labelGreaterThanOrEqual);
```

Если `iSrc` пройдет проверку, появится возможность доступа к массиву источника. При выполнении кода операции `Ldelem` предполагается, что сам массив и индекс массива уже присутствуют в стеке. Эти два значения извлекаются и заменяются элементом массива, имеющим этот индекс. Элемент этого кода операции `U1` определяет, что элемент массива является беззнаковым однобайтовым значением:

```
generator.Emit(OpCodes.Ldelem_U1);
generator.Emit(OpCodes.Conv_R8);
```

Код операции `Conv_R8` преобразует значение в стеке в восьмибайтовое значение с плавающей запятой и возвращает его обратно в стек.

Теперь байт из `iSrc` находится в стеке и уже преобразован в число с плавающей запятой. Он готов для перемножения с элементом фильтра. Поскольку на момент генерации метода значение элемента фильтра уже известно, код C#

пропускает операцию умножения, если элемент фильтра содержит единицу (для элемента фильтра со значением 1 умножение не требуется):

```
if (filter[iFilter] == 1)
{
    // элемент src в стеке, поэтому ничего делать не нужно
}
```

Если элемент фильтра содержит -1, байт источника может быть просто инвертирован, возможно, сохраняя при этом часть процессорного времени по сравнению с умножением:

```
else if (filter[iFilter] == -1)
{
    generator.Emit(OpCodes.Neg);
}
```

В противном случае байт перемножается с элементом фильтра:

```
else
{
    generator.Emit(OpCodes.Ldc_R8, filter[iFilter]);
    generator.Emit(OpCodes.Mul);
}
```

Вы, возможно, вспомните, что `pixelsAccum` был определен как локальная переменная с индексом 1. Следующий фрагмент кода помещает `pixelsAccum` в стек, прибавляет к нему байт источника, перемноженный с элементом фильтра, и помещает результат обратно в `pixelsAccum`:

```
generator.Emit(OpCodes.Ldloc_1);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Stloc_1);
```

Подобным же образом `filterAccum` (имеющий индекс локальной переменной, равный 2) должен аккумулировать значения элементов фильтра:

```
generator.Emit(OpCodes.Ldc_R8, filter[iFilter]);
generator.Emit(OpCodes.Ldloc_2);
generator.Emit(OpCodes.Add);
generator.Emit(OpCodes.Stloc_2);
generator.Emit(OpCodes.Br, labelLoopBottom);
```

Теперь мы оказались в нижней части внутреннего цикла `for`, что соответствует строке 24 примера 8.2. По сути мы завершили обработку каждого элемента фильтра, за исключением того, что стек должен быть очищен в расчете на те случаи, когда вычисляемый индекс `iSrc` выходит за границы изображения. Эта часть генерируемого кода (в C# – в нижней части цикла `for` для `iFilter`) устанавливает три метки и осуществляет очистку путем выталкивания неиспользуемых элементов из стека:

```
generator.MarkLabel(labelLessThanZero);
generator.Emit(OpCodes.Pop);
generator.MarkLabel(labelGreaterThanOrEqual);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Pop);
generator.MarkLabel(labelLoopBottom);
}
```

Теперь для конкретного целевого пикселя сгенерирован весь код, вычисляющий `pixelsAccum` и `filterAccum`. Результат уже почти готов к переносу в массив `dst`. В массив загружаются ссылка на массив (имеющая индекс аргумента метода, равный 1) и индекс `iDst` (имеющий индекс локальной переменной, равный 0):

```
generator.Emit(OpCodes.Ldarg_1); // массив dst
generator.Emit(OpCodes.Ldloc_0); // индекс iDst
```

Здесь могут быть задействованы некоторые переходы, поэтому определяются следующие метки:

```
Label labelSkipDivide = generator.DefineLabel();
Label labelCopyQuotient = generator.DefineLabel();
Label labelBlack = generator.DefineLabel();
Label labelWhite = generator.DefineLabel();
Label labelDone = generator.DefineLabel();
```

В следующем фрагменте кода для подготовки к осуществлению операции деления в стек загружаются локальные переменные `pixelsAccum` и `filterAccum`. Сначала путем сравнения `filterAccum` с нулем нужно проверить, не будет ли деления на нуль. Этот код эквивалентен строке 25 примера 8.2:

```
generator.Emit(OpCodes.Ldloc_1);           // pixelsAccum
generator.Emit(OpCodes.Ldloc_2);           // filterAccum
generator.Emit(OpCodes.Dup);               // Изготовление копии
generator.Emit(OpCodes.Ldc_R8, 0.0);        // Помещение 0 в стек
generator.Emit(OpCodes.Beq_S, labelSkipDivide);
```

Если знаменатель не равен нулю, выполняется деление, и в стеке остается частное от него:

```
generator.Emit(OpCodes.Div);
generator.Emit(OpCodes.Br_S, labelCopyQuotient);
```

Если `filterAccum` равен нулю, выполняется следующий код, и исходный экземпляр `filterAccum` выталкивается из стека:

```
generator.MarkLabel(labelSkipDivide);
generator.Emit(OpCodes.Pop);                // Выталкивание filterAccum
```

В любом случае в стеке остается `pixelsAccum`, либо поделенный на `filterAccum`, либо нет. С этого частного делаются две копии:

```
generator.MarkLabel(labelCopyQuotient);
generator.Emit(OpCodes.Dup);                // Изготовление копии частного
generator.Emit(OpCodes.Dup);                // Изготовление еще одной копии
```

Большая часть из того, что следует дальше, является кодом Intermediate Language, эквивалентным оператору строк 28 и 29 примера 8.2. Если частное от деления меньше нуля, в коде осуществляется переход на метку, где значению целевого пикселя должен быть присвоен 0:

```
generator.Emit(OpCodes.Ldc_R8, 0.0);
generator.Emit(OpCodes.Bit_S, labelBlack);
```

Если частное больше 255, то следующий код осуществляет переход на метку, где значению целевого пикселя должно быть присвоено число 255:

```
generator.Emit(OpCodes.Ldc_R8, 255.0);
generator.Emit(OpCodes.Bgt_S, labelWhite);
```

В противном случае значение, находящееся в стеке, преобразуется в беззнаковое однобайтовое значение:

```
generator.Emit(OpCodes.Conv_U1);
generator.Emit(OpCodes.Br_S, labelDone);
```

Следующий фрагмент кода предназначен для тех случаев, когда в целевой массив должен быть помещен нулевой байт. Команда Ldc_I4_S помещает в стек однобайтовое значение, но оно попадает в стек в формате четырехбайтового целого числа, поскольку ширина области памяти стека имеет четырехбайтовое приращение:

```
generator.MarkLabel(labelBlack);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Ldc_I4_S, 0);
generator.Emit(OpCodes.Br_S, labelDone);
```

Эта часть кода подобна той, в которой число 255 должно быть сохранено в целевом массиве:

```
generator.MarkLabel(labelWhite);
generator.Emit(OpCodes.Pop);
generator.Emit(OpCodes.Ldc_I4_S, 255);
```

И теперь мы наконец-то готовы для сохранения байта в целевом массиве. Массив dst уже в стеке, также в стеке и индекс iDst, и значение, которое должно быть сохранено в массиве. Команда Stelem_Il сохраняет однобайтовое значение в массиве:

```
generator.MarkLabel(labelDone);
generator.Emit(OpCodes.Stelem_Il);
```

Теперь мы уже достигли нижней части цикла iDst, что эквивалентно строке 30 примера 8.2. И сейчас значение локальной переменной iDst должно быть увеличено на единицу, и это значение нужно сравнить с количеством байтов в массиве. Если оно меньше, код осуществляет переход на вершину цикла:

```
generator.Emit(OpCodes.Ldloc_0);           // Помещение iDst в стек
generator.Emit(OpCodes.Ldc_I4_1);           // Помещение в стек числа 1
generator.Emit(OpCodes.Add);                // Увеличение iDst на единицу
generator.Emit(OpCodes.Dup);                // Изготовление копии
generator.Emit(OpCodes.Stloc_0);             // Сохранение результата в iDst
generator.Emit(OpCodes.Ldc_I4_cBytes);       // Помещение в стек
                                           // значения cBytes
generator.Emit(OpCodes.Blt, labelTop);       // Переход в начало, если
                                           // iDst < cBytes
```

После завершения цикла сгенерированный метод заканчивается кодом возврата:

```
generator.Emit(OpCodes.Ret);
```

Теперь код Intermediate Language уже сгенерирован. Экземпляр метода DynamicMethod, созданный в начале метода FilterMethodIL, завершен и готов к выполнению, или к вызову (invoke), что подразумевается в имени следующего метода. Второй аргумент метода Invoke определяет два аргумента генерируемого метода как массивы src и dst:

```
} dynameth.Invoke(this, new object[] { src, dst });
```

И на этом метод FilterMethodIL завершается. Теперь объекты DynamicMethod и ILGenerator находятся вне области видимости, и та часть памяти, которая ими занята, может быть освобождена имеющимся в .NET сборщиком мусора.

Алгоритмы, реализованные на языках низкого уровня, обычно работают быстрее, чем те, которые реализованы на высококоуровневых языках, а узкоспециализированные алгоритмы всегда работают быстрее, чем универсальные. Похоже, путем динамической специализации алгоритма на Intermediate Language непосредственно перед его применением мы убиваем сразу двух зайцев. Алгоритм универсален до тех пор, пока он не должен быть специализирован, а затем он проходит специализацию с применением более эффективного кода.

Негативная сторона этого процесса заключается в том, что вам нужно становиться своеобразным создателем компиляторов и ломать барьер между кодом и данными, входя, таким образом, в некую преисподнюю, где код и данные становятся зеркальным отображением друг друга.

Безусловно, создание метода FilterMethodIL потребовало большого объема работы, но зато посмотрите, насколько хорошо решены проблемы его производительности. В общем, FilterMethodIL, генерирующий на лету команды Intermediate Language, работает в четыре раза быстрее, и он в некотором смысле лучше, чем метод FilterMethodCS, реализованный непосредственно на C#.

Вы, конечно, можете посчитать код FilterMethodIL уродливым, и я готов признать, что это не самый красивый из встречавшихся мне примеров кода. Но когда алгоритм начинает работать вчетверо быстрее, чем работал в предыдущей реализации, тогда единственным подходящим для этого словом может быть только *красота*.

9

Нисходящая иерархия операторов

Дуглас Крокфорд (Douglas Crockford)

В 1973 году Боган Пратт (Vaughan Pratt) представил статью «Нисходящая иерархия операторов»¹ на первом ежегодном симпозиуме по основам языков программирования (Principles of Programming Languages Symposium) в Бостоне. В этой статье Пратт описал методику синтаксического анализа, сочетающую лучшие свойства метода рекурсивного спуска и метода приоритета операторов Роберта Флойда (Robert W Floyd)². Он утверждал, что методика отличается простотой восприятия, легкостью реализации, удобством, исключительной эффективностью и невероятной гибкостью. Я бы добавил, что она еще и красива.

Может показаться странным, что таким идеальным подходом к конструкции компилятора сегодня полностью пренебрегают. Почему так получилось? В своей статье Пратт предположил, что пристрастие к грамматике форм Бэкуса–Наура (BNF) и ее различным потомкам, наряду со связанными с ней автоматами и теоремами, воспрепятствовало развитию в тех направлениях, которые не проглядываются в сфере теории автоматов.

Другое объяснение состоит в том, что его методика наиболее эффективна при использовании в динамичном, функциональном языке программирования. Ее использование в статичном, процедурном языке было бы значительно затруднено. В своей статье Пратт пользовался языком LISP и практически без всяких усилий выстраивал дерево анализа из потока лексем.

Но техника синтаксического анализа не слишком ценилась в LISP-сообществе, которое славилось спартанским отрицанием синтаксиса. Со временем создания LISP было предпринято немало попыток придать языку более богатый, ALGOL-подобный синтаксис, включая:

¹ Статья Пратта доступна по адресу: <http://portal.acm.org/citation.cfm?id=512931>; дополнительные сведения о самом Пратте можно найти по адресу: <http://boole.stanford.edu/pratt.html>.

² Чтобы узнать о Флойде, посмотрите статью «Robert W Floyd, In Memoriam» Дональда Кнута по адресу: <http://sigact.acm.org/floyd>.

принадлежащий Пратту CGOL

<http://zane.brouhaha.com/~healyzh/doc/cgol.doc.txt;>

LISP 2

http://community.computerhistory.org/scc/projects/LISP/index.html#LISP_2_;

MLISP

<ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/68/92/CS-TR-68-92.pdf;>

Dylan

<http://www.opendylan.org;>

Clisp, принадлежащий Interlisp,

<http://community.computerhistory.org/scc/projects/LISP/interlisp/Teitelman-3IJCAI.pdf.>

Оригинальные метавыражения – M-expressions, принадлежащие МакКарти (McCarthy)

<http://www-formal.stanford.edu/jmc/history/lisp/lisp.html>

Но ни один из этих синтаксисов так и не прижился. Сообщество функционального программирования пришло к заключению, что соответствие между программами и данными имеет куда большее значение, чем выразительный синтаксис.

Но основному программистскому сообществу нравится предлагаемый синтаксис, поэтому LISP никогда им не признавался. Методика Пратта вполне подходит динамичным языкам, но сообщество их приверженцев исторически не воспринимало тот синтаксис, который реализовывался этой методикой.

JavaScript

Ситуация изменилась с появлением JavaScript, динамичного, функционального языка, но по синтаксису, безусловно, принадлежащего к семейству языков Си. JavaScript – динамичный язык, сторонникам которого нравится синтаксис, вдобавок ко всему он еще и объектно-ориентированный язык. Статья, написанная Праттом в 1973 году, предвосхитила объектную ориентацию, но для нее в этой статье не хватало выразительной нотации. А JavaScript обладает ярко выраженной объектной нотацией. Поэтому JavaScript оказался идеальным языком для применения методики Пратта. Я покажу, как в JavaScript можно быстро и без особых усилий создавать синтаксические анализаторы.

В такой краткой главе подробно рассмотреть язык JavaScript просто невозможно, да, наверное, этого и не следует делать, чтобы не вносить лишнюю путаницу. Но в этом языке есть ряд превосходных, достойных рассмотрения вещей. Мы создадим синтаксический анализатор, который сможет вести обработку упрощенного JavaScript, и напишем этот анализатор на этой же упрощенной версии языка. Упрощенный JavaScript – вполне достойное нашего внимания средство, включающее следующее.

Функции как объекты первого класса

В лексической сфере функции — это начала координат.

Динамические объекты с наследованием прототипов

Объекты, не связанные с классами. Мы можем добавить новый элемент к любому объекту обычным присваиванием. Объект может унаследовать элементы от другого объекта.

Объектные литералы и литералы массивов

Это очень удобная система записи для создания новых объектов и массивов. Литералы JavaScript стимулировали возникновение формата обмена данными JSON (<http://www.JSON.org>).

Мы воспользуемся заложенной в JavaScript особенностью работы с прототипами для создания лексических объектов, унаследованных от обозначений, и обозначений, унаследованных от исходных обозначений. Основанием для этого послужит функция `Object`, создающая новый объект, наследующий элементы уже существующего объекта. Реализация нашего замысла также будет основана на построителе лексем (tokenizer), превращающем строку в массив простых лексем. В процессе построения дерева синтаксического анализа мы будем производить перебор элементов массива лексем.

Таблица обозначений

Для управления синтаксическим анализатором мы воспользуемся таблицей обозначений:

```
var symbol_table = {};
```

Объект `original_symbol` послужит прототипом для всех остальных обозначений. В нем содержатся методы, сообщающие об ошибках. Как правило, им на замену приходят более полезные методы:

```
var original_symbol = {
    nud: function () {
        this.error("Неопределен.");
    },
    led: function (left) {
        this.error("Пропущен оператор.");
    }
};
```

Давайте рассмотрим функцию, которая определяет обозначения. Она принимает идентификатор — `id` и, дополнительно, степень его связанности, которая по умолчанию равна нулю. Возвращает она объект обозначения, соответствующий указанному `id`. Если обозначение уже присутствует в `symbol_table`, то возвращается его объект. Иначе функция создает новый объект обозначения, унаследованный из `original_symbol`, сохраняет его в таблице обозначений и возвращает этот объект. Изначально объект обозначений содержит `id`, свое значение, переданную ему степень связанности (`lbp`) и все, что он унаследовал от `original_symbol`:

```

var symbol = function (id, bp) {
    var s = symbol_table[id];
    bp = bp || 0;
    if (s) {
        if (bp >= s.lbp) {
            s.lbp = bp;
        }
    } else {
        s = object(original_symbol);
        s.id = s.value = id;
        s.lbp = bp;
        symbol_table[id] = s;
    }
    return s;
};

```

Следующие обозначения относятся к широко распространенным разделителям и закрывающим элементам:

```

symbol(":");
symbol(":");
symbol(".");
symbol(")");
symbol("]");
symbol("}");
symbol("else");

```

Обозначение (end) указывает на то, что лексем больше нет. А обозначение (name) служит прототипом для новых имен, к примеру, для имен переменных. Столь необычная запись выбрана для того, чтобы избежать коллизий:

```

symbol("(end)");
symbol("(name)");

```

Обозначение (literal) служит прототипом для всех строковых и числовых литералов:

```

var itself = function () {
    return this;
};
symbol("(literal)").nud = itself;

```

Обозначение this является специальной переменной. При вызове метода оно дает ссылку на объект:

```

symbol("this").nud = function () {
    scope.reserve(this);
    this.arity = "this";
    return this;
};

```

Лексемы

Допустим, что исходный текст был преобразован в массив простейших лексических объектов (tokens), каждый из которых содержит элемент типа — type, являющийся строкой ("name", "string", "number", "operator"), и элемент значения — value, который является строкой или числом.

Переменная `token` всегда содержит текущую лексему:

```
var token;
```

Функция `advance` создает новый лексический объект и присваивает его переменной `token`. Эта функция принимает дополнительный параметр `id`, который она может сравнивать с `id` предыдущей лексемы. Новый прототип лексического объекта должен быть поименованной лексемой в текущем пространстве имен или обозначением из таблицы обозначений. Для новой лексемы свойство `arity` будет принимать значения `"name"`, `"literal"` или `"operator"`. Позже, когда роль лексемы в программе станет более понятной, это свойство может быть изменено на `"binary"`, `"unary"` или `"statement"`:

```
var advance = function (id) {
    var a, o, t, v;
    if (id && token.id !== id) {
        token.error("Ожидался '" + id + "'.");
    }
    if (token_nr >= tokens.length) {
        token = symbol_table["(end)"];
        return;
    }
    t = tokens[token_nr];
    token_nr += 1;
    v = t.value;
    a = t.type;
    if (a === "name") {
        o = scope.find(v);
    } else if (a === "operator") {
        o = symbol_table[v];
        if (!o) {
            t.error("Неизвестный оператор.");
        }
    } else if (a === "string" || a === "number") {
        a = "literal";
        o = symbol_table["(literal)"];
    } else {
        t.error("Неизвестная лексема.");
    }
    token = object(o);
    token.value = v;
    token.arity = a;
    return token;
};
```

Старшинство

Лексемы — это объекты, обладающие методами, позволяющими им выбирать старшинство, соответствовать другим лексемам и выстраивать деревья (а в более амбициозных проектах к тому же осуществлять проверку типов, оптимизацию и генерацию кода). Основная проблема старшинства заключается в следующем вопросе: с каким оператором связан заданный операнд, находящийся

между двумя операторами, с левым или с правым? Если в следующем выражении A и B являются операторами:

$d A e B f$

то с каким из них связан operand e : с A или с B ? Другими словами, о какой схеме идет речь, об этой:

$(d A e) B f$

или об этой:

$d A (e B f)$

В конечном счете, сложность процесса синтаксического анализа сводится к разрешению этой неоднозначности. Разрабатываемая здесь методика использует объекты лексем, элементы которых включают степень связанности (или уровни старшинства) и простые методы под названием *nud* (null denotation — нулевое обозначение) и *led* (left denotation — левое обозначение). *nud*, в отличие от *led*, не касается лексем, расположенных слева, и используется значениями (такими как переменные и литералы) и префиксными операторами. Метод *led* используется инфиксными (бинарными) и суффиксными операторами. Лексема может иметь оба метода, и *nud* и *led*. Например, лексема $-$ может быть как префиксным оператором (отрицание), так и инфиксным оператором (вычитание), поэтому у него будут оба метода.

Выражения

Ядром методики Пратта является функция выражения — *expression*. Она принимает правую степень связанности, управляющую агрессивностью ее потребления находящихся справа лексем, и возвращает результат вызова методов тех лексем, на которые она воздействует:

```
var expression = function (rbp) {
    var left;
    var t = token;
    advance();
    left = t.nud();
    while (rbp < token.lbp) {
        t = token;
        advance();
        left = t.led(left);
    }
    return left;
}
```

Метод *expression* вызывает метод *nud* лексемы *token*. Метод *nud* используется для обработки литералов, переменных и префиксных операторов. После этого, до тех пор пока правая степень связанности меньше, чем левая степень связанности следующей лексемы, вызываются методы *led*. Метод *led* используется для обработки инфиксных и суффиксных операторов. Этот процесс может носить рекурсивный характер, поскольку методы *nud* и *led* могут вызывать метод *expression*.

Инфиксные операторы

Оператор + относится к инфиксным операторам, поэтому в относящемся к нему объекте будет содержаться метод led, который превращает объект лексемы в дерево, двумя ветвями которого являются операнды слева и справа от +. Левый операнд передается в led, а правый берется за счет вызова метода expression.

Число 60 выражает степень связанности обозначения +. Операторы, имеющие более сильную связь или являющиеся выше по старшинству, имеют более высокие показатели степени связанности. В ходе превращения потока лексем в дерево синтаксического разбора мы будем использовать лексемы оператора в качестве контейнеров узлов операндов:

```
symbol("+", 60).led = function (left) {
    this.first = left;
    this.second = expression(60);
    this.arity = "binary";
    return this;
};
```

При определении обозначения для оператора * мы видим, что оно отличается лишь идентификатором id и степенью связанности. Поскольку оно имеет более сильную связь, его показатель степени связанности имеет более высокое значение:

```
symbol("*", 70).led = function (left) {
    this.first = left;
    this.second = expression(70);
    this.arity = "binary";
    return this;
};
```

На них будут похожи многие, хотя и не все инфиксные операторы, поэтому мы можем упростить себе работу, определив функцию infix, которая поможет определять инфиксные операторы. Эта функция будет принимать id и показатель степени связанности и дополнительно — функцию led. Если функция led предоставляться не будет, то она будет добавлена по умолчанию, что в большинстве случаев может оказаться полезным:

```
var infix = function (id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp);
        this.arity = "binary";
        return this;
    };
    return s;
}
```

Это позволит придать объявлению операторов более наглядный стиль:

```
infix("+", 60);
infix("-", 60);
infix("*", 70);
infix("/", 70);
```

Строка === представляет в JavaScript оператор строгого сравнения:

```
infix("==", 50);
infix("!=" , 50);
infix("<" , 50);
infix("<=" , 50);
infix(">" , 50);
infix(">=" , 50);
```

Трехэлементный (тернарный) оператор принимает три выражения, разделенные знаками ? и : . Его неординарность заставляет нас предоставить определению функцию led:

```
infix("?", 20, function (left) {
    this.first = left;
    this.second = expression(0);
    advance(":");
    this.third = expression(0);
    this.arity = "ternary";
    return this;
});
```

Оператор . используется для выбора элемента объекта. Лексема справа должна быть именем, но она будет использована в качестве литерала:

```
infix(".", 90, function (left) {
    this.first = left;
    if (token.arity !== "name") {
        token.error("Ожидалось имя свойства.");
    }
    token.arity = "literal";
    this.second = token;
    this.arity = "binary";
    advance();
    return this;
});
```

Оператор [используется для динамического выбора элемента из объекта или массива. Находящееся справа выражение должно закрываться символом]:

```
infix("[", 90, function (left) {
    this.first = left;
    this.second = expression(0);
    this.arity = "binary";
    advance("]");
    return this;
});
```

Все эти инфиксные операторы имеют левостороннюю ассоциацию. Уменьшая степень связности справа, можно создать и операторы с правосторонней ассоциацией, такие как логические шунтирующие операторы:

```
var infixr = function (id, bp, led) {
    var s = symbol(id, bp);
    s.led = led || function (left) {
        this.first = left;
        this.second = expression(bp - 1);
        this.arity = "binary";
        return this;
    };
    return s;
}
```

Оператор `&&` возвращает первый operand, если его значение ложно. Иначе он возвращает второй operand. Оператор `||` возвращает первый operand, если его значение истинно; иначе он возвращает второй operand:

```
infixr("&&", 40);
infixr("||", 40);
```

Префиксные операторы

Для префиксных операторов можно сделать то же самое. Префиксные операторы имеют правостороннюю ассоциацию. У префиксов отсутствует показатель степени связности слева, потому что слева от него нет ничего, что было бы с ним связано. Префиксные операторы иногда могут быть зарезервированными словами (которые рассматриваются далее в разделе «Область видимости»):

```
var prefix = function (id, nud) {
    var s = symbol(id);
    s.nud = nud || function () {
        scope.reserve(this);
        this.first = expression(80);
        this.arity = "unary";
        return this;
    };
    return s;
}
prefix("-");
prefix("!");
prefix("typeof");
```

Метод `nud` обозначения (будет вызывать функцию `advance("`), чтобы сопоставить ему парную лексему). Лексема (не становится частью дерева синтаксического анализа, поскольку `nud` возвращает выражение:

```
prefix(".").function () {
    var e = expression(0);
    advance(")");
    return e;
});
```

Операторы присваивания

Для определения операторов присваивания можно воспользоваться функцией `infixr`, но тут нужно решить еще две небольшие задачи, поэтому мы создадим специализированную функцию `assignment`. Она будет проверять левый operand на приемлемость левой части значения. Кроме этого, мы установим флагок `assignment`, чтобы в последующем операторы присваивания можно было быстро идентифицировать:

```
var assignment = function (id) {
    return infixr(id, 10, function (left) {
        if (left.id !== "." && left.id !== "[" &&
            left.arity !== "name") {
```

```

        left.error("Неприемлемая левая часть значения.");
    }
    this.first = left;
    this.second = expression(9);
    this.assignment = true;
    this.arity = "binary";
    return this;
}:
assignment("=");
assignment("+=");
assignment("-=");

```

Обратите внимание, что мы воспользовались своеобразной схемой наследования, где функция `assignment` возвращает результат вызова функции `infixr`, а `infixr` возвращает результат вызова функции `symbol`.

Константы

Функция `constant` встраивает в язык константы. Метод `nud` видоизменяет поименованную лексему в лiteralную:

```

var constant = function (s, v) {
    var x = symbol(s);
    x.nud = function () {
        scope.reserve(this);
        this.value = symbol_table[this.id].value;
        this.arity = "literal";
        return this;
    };
    x.value = v;
    return x;
};
constant("true", true);
constant("false", false);
constant("null", null);
constant("pi", 3.141592653589793);

```

Область видимости

Для определения применяемых в языке обозначений мы пользуемся такими функциями, как `infix` и `prefix`. Большинство языков для определения таких новых обозначений, как имена переменных, имеют соответствующую систему записи. В самых простых языках при обнаружении нового слова мы можем дать ему определение и занести в таблицу обозначений. В более сложных языках может потребоваться описание области видимости, дающее программисту удобное средство управления жизненным циклом и видимостью переменной.

Область видимости – это пространство программы, в котором переменная определена и доступна. Области видимости могут вкладываться внутрь других

областей видимости. Переменные, определенные внутри какой-нибудь области видимости, невидимы за ее пределами.

Мы сохраним текущий объект области видимости в переменной scope:

```
var scope;
```

Объект original_scope является прототипом для всех объектов области видимости. Он содержит метод define, который используется для определения новой переменной в области видимости. Метод define превращает лексему имени в лексему переменной. Если переменная уже была определена в области видимости или имя уже было использовано в качестве зарезервированного слова, он выдает ошибку:

```
var original_scope = {
    define: function (n) {
        var t = this.def[n.value];
        if (typeof t === "object") {
            n.error(t.reserved ?
                "Имя зарезервировано." :
                "Имя уже определено.");
        }
        this.def[n.value] = n;
        n.reserved = false;
        n.nud = itself;
        n.led = null;
        n.std = null;
        n.lbp = 0;
        n.scope = scope;
        return n;
    }
}.
```

Метод find используется для поиска определения имени. Он начинает искать в текущей области видимости и в случае необходимости проходит всю цепочку родительских областей видимости, заканчивая поиск в таблице обозначений. Если он не может найти определение, то возвращает symbol_table[`"(name)"`]:

```
find: function (n) {
    var e = this;
    while (true) {
        var o = e.def[n];
        if (o) {
            return o;
        }
        e = e.parent;
        if (!e) {
            return symbol_table[
                symbol_table.hasOwnProperty(n) ?
                n : "(name)"];
        }
    }
}.
```

Метод pop закрывает область видимости:

```
pop: function () {
    scope = this.parent;
}.
```

Метод `reserve` используется для обозначения, что данное имя задействовано в данной области видимости в качестве зарезервированного слова:

```
reserve: function (n) {
  if (n.arity !== "name" || n.reserved) {
    return;
  }
  var t = this.def[n.value];
  if (t) {
    if (t.reserved) {
      return;
    }
    if (t.arity === "name") {
      n.error("Имя уже определено.");
    }
  }
  this.def[n.value] = n;
  n.reserved = true;
}
};
```

Для зарезервированных слов нам нужна определенная установка. В некоторых языках те слова, которые задействованы в структуре (например `if`), являются зарезервированными и не могут быть использованы в качестве имен переменных. Гибкость нашего синтаксического анализатора позволяет воспользоваться более практической установкой. Например, мы можем сказать, что в любой функции любое имя может быть использовано в качестве структурного слова или в качестве переменной, но только не в обоих качествах сразу. Кроме того, локальное резервирование слов происходит только после того, как они уже задействованы в качестве зарезервированных слов. Это улучшает ситуацию для разработчика языка, поскольку добавление к языку новых структурных слов не приведет к нарушениям в существующих программах, а также упрощает жизнь программистам, поскольку они не испытывают стеснений из-за каких-то нелепых ограничений в отношении использования имен.

Когда нужно установить для функции или блока новую область видимости, мы вызываем функцию `new_scope`, которая создает новый экземпляр исходного прототипа области видимости:

```
var new_scope = function () {
  var s = scope;
  scope = object(original_scope);
  scope.def = {};
  scope.parent = s;
  return scope;
};
```

Операторы

Оригинальная формулировка Пратта работала с функциональными языками, в которых все построено на выражениях. Операторы имеются в большинстве основных языков. Добавление к лексемам еще одного, предназначенного для

определения операторов метода — std (statement denotation), поможет легко с ними справиться. Метод std похож на метод nud, за исключением того, что он используется только в начале оператора.

Функция statement производит синтаксический анализ одного оператора. Если текущая лексема имеет метод std, она резервируется, и вызывается метод std. В противном случае мы предполагаем, что оператор-выражение завершается символом .. Для надежности мы отклоняем оператор-выражение, который не является присваиванием или вызовом:

```
var statement = function () {
    var n = token.v;
    if (n.std) {
        advance();
        scope.reserve(n);
        return n.std();
    }
    v = expression(0);
    if (!v.assignment && v.id !== "(") {
        v.error("Неверный оператор-выражение.");
    }
    advance(":");
    return v;
}:
```

Функция statements производит синтаксический анализ операторов до тех пор, пока не встретит (end) или }, которые сигнализируют об окончании блока. Она возвращает оператор, массив операторов или (если операторы отсутствовали) просто null:

```
var statements = function () {
    var a = [], s;
    while (true) {
        if (token.id === ")" || token.id === "(end)") {
            break;
        }
        s = statement();
        if (s) {
            a.push(s);
        }
    }
    return a.length === 0 ? null : a.length === 1 ? a[0] : a;
}:
```

Функция stmt используется для добавления операторов к таблице обозначений. Она принимает id оператора и функцию std:

```
var stmt = function (s, f) {
    var x = symbol(s);
    x.std = f;
    return x;
}:
```

Оператор блока заключает список операторов в пару фигурных скобок, придавая им новую область видимости:

```
stmt("{", function () {
    new_scope();
```

```

var a = statements( );
advance("}"):
scope.pop( ):
return a;
}):

```

Функция block проводит синтаксический анализ блока:

```

var block = function ( ) {
  var t = token;
  advance("{");
  return t.std( );
}:

```

Оператор **var** определяет одну или несколько переменных в текущем блоке.
За каждым именем может дополнительно следовать символ **=** и выражение:

```

stmt("var", function ( ) {
  var a = [], n, t;
  while (true) {
    n = token;
    if (n.arity !== "name") {
      n.error("Ожидалось новое имя переменной.");
    }
    scope.define(n);
    advance( );
    if (token.id === "=") {
      t = token;
      advance("=");
      t.first = n;
      t.second = expression(0);
      t.arity = "binary";
      a.push(t);
    }
    if (token.id === ".") {
      break;
    }
    advance(".");
  }
  advance(":");
  return a.length === 0 ? null : a.length === 1 ? a[0] : a;
}):

```

Оператор **while** определяет цикл. Он содержит выражение, заключенное в круглые скобки, и блок:

```

stmt("while", function ( ) {
  advance("(");
  this.first = expression(0);
  advance(")");
  this.second = block( );
  this.arity = "statement";
  return this;
}):

```

Оператор **if** позволяет осуществлять условное выполнение. Если после блока встречается обозначение **else**, мы производим синтаксический анализ следующего блока или оператора **if**:

```

stmt("if", function () {
    advance("(");
    this.first = expression(0);
    advance(")");
    this.second = block();
    if (token.id === "else") {
        scope.reserve(token);
        advance("else");
        this.third = token.id === "if" ? statement() : block();
    }
    this.arity = "statement";
    return this;
}):

```

Оператор `break` используется для прерывания цикла. Мы удостоверяемся, что за ним следует обозначение `:`:

```

stmt("break", function () {
    advance(":");
    if (token.id !== ":") {
        token.error("Недоступный оператор.");
    }
    this.arity = "statement";
    return this;
}):

```

Оператор `return` используется для возврата из функций. Он может возвращать необязательное выражение:

```

stmt("return", function () {
    if (token.id !== ":") {
        this.first = expression(0);
    }
    advance(":");
    if (token.id !== ")") {
        token.error("Недоступный оператор.");
    }
    this.arity = "statement";
    return this;
}):

```

ФУНКЦИИ

Функции являются выполняемыми объектными величинами. У функции есть произвольное имя (и она может вызывать саму себя рекурсивно), перечень имен параметров, заключенный в круглые скобки. Функция обладает собственной областью видения:

```

prefix("function", function () {
    var a = [];
    scope = new_scope();
    if (token.arity === "name") {
        scope.define(token);
        this.name = token.value;
        advance();
    }
})

```

```

advance("(");
if (token.id !== ")") {
    while (true) {
        if (token.arity !== "name") {
            token.error("Ожидалось имя параметра.");
        }
        scope.define(token);
        a.push(token);
        advance( );
        if (token.id !== ".") {
            break;
        }
        advance(".");
    }
}
this.first = a;
advance(")");
advance("{");
this.second = statements( );
advance("}");
this.arity = "function";
scope.pop( );
return this;
}):

```

Функции вызываются оператором `.`. Он может принимать от нуля и более аргументов, разделенных запятыми. Чтобы определить выражения, которые не могут быть значениями функции, мы смотрим на левый operand:

```

infix("( . 90. function (left) {
    var a = [];
    this.first = left;
    this.second = a;
    this.arity = "binary";
    if ((left.arity !== "unary" ||
        left.id !== "function") &&
        left.arity !== "name" &&
        (left.arity !== "binary" ||
        (left.id !== "." &&
        left.id !== "(" &&
        left.id !== "["))) {
        left.error("Ожидалось имя переменной.");
    }
    if (token.id !== ")") {
        while (true) {
            a.push(expression());
            if (token.id !== ".") {
                break;
            }
            advance(".");
        }
    }
    advance(")");
    return this;
}):
```

Литералы массивов и объектов

Литерал массива — это набор квадратных скобок, окружающих нуль или несколько выражений, разделенных запятыми. Каждое из выражений вычисляется, а результаты собираются в новом массиве:

```
prefix("[", function () {
    var a = [];
    if (token.id !== "]") {
        while (true) {
            a.push(expression(0));
            if (token.id === ",") {
                break;
            }
            advance(".");
        }
    }
    advance("]"):
    this.first = a;
    this.arity = "unary";
    return this;
}):
```

Литерал объекта — это набор фигурных скобок, окружающих нуль или несколько пар, разделенных запятыми. Под *парами* понимаются пары *ключ/выражение*, разделенные символом `:`. Ключ представляет собой литерал или имя, рассматриваемое в качестве литерала:

```
prefix("{", function () {
    var a = [];
    if (token.id !== "}") {
        while (true) {
            var n = token;
            if (n.arity !== "name" && n.arity !== "literal") {
                token.error("Неверный ключ.");
            }
            advance( );
            advance(":");
            var v = expression(0);
            v.key = n.value;
            a.push(v);
            if (token.id === ",") {
                break;
            }
            advance(".");
        }
    }
    advance("}"):
    this.first = a;
    this.arity = "unary";
    return this;
}):
```

Что нужно сделать и о чем подумать

Простой синтаксический анализатор, показанный в этой главе, легко поддается расширению. Дерево может быть передано генератору кода или интерпретатору. Для создания дерева требуются весьма незначительные вычисления. И как мы уже выяснили, чтобы написать программу, которая выстраивает это дерево, требуются крайне незначительные усилия.

Мы могли бы создать функцию `infix`, принимающую код операции, который можно использовать при генерации кода. Мы могли бы также заставить ее принять дополнительные методы, которые могут быть использованы для постоянного свертывания и генерации кода.

Мы могли бы добавить дополнительные операторы, к примеру, `for`, `switch` и т.д. Мы могли бы добавить метки операторов, добавить дополнительные проверки на наличие ошибок и ввести систему их устранения. Мы могли бы добавить значительно большее количество операторов, определений типов и логических выводов. Можно было бы сделать наш язык расширяемым. С той же легкостью, с которой мы можем определять новые переменные, мы могли бы позволить программисту добавлять новые операторы и инструкции.

Вы можете испытать демонстрационную версию синтаксического анализатора, рассмотренного в этой главе, обратившись по адресу: <http://javascript.crockford.com/tdop/index.html>.

Другой пример этой методики синтаксического анализа можно найти в JSLint по адресу: <http://JSLint.com>.

10 Поиск методов ускоренного подсчета заполнения

Генри Уоррен-мл. (Henry S. Warren, Jr.)

Подсчет заполнения, или нахождение продольной суммы, относится к фундаментальным и обманчиво простым компьютерным алгоритмам, вычисляющим количество единичных битов в компьютерном слове. Функция подсчета заполнения находила применение при решении разных задач, от самых простых до самых сложных. К примеру, если последовательности представлены битовыми строками, подсчет заполнения дает размер последовательности. Функция также может быть использована для генерации биноминально распределенной случайной целочисленной величины. И это, и другие ее применения рассматриваются в конце данной главы.

Хотя применение этой операции является не самым распространенным явлением, многие компьютеры — преимущественно те, которые в свое время считались суперкомпьютерами, — имели для нее отдельную команду. К этим компьютерам можно отнести Ferranti Mark I (1951), IBM Stretch computer (1960), CDC 6600 (1964), построенную в России БЭСМ-6 (1967), Cray 1 (1976), Sun SPARCv9 (1994) и IBM Power 5 (2004).

В этой главе рассматриваются способы подсчета заполнения на тех машинах, которые не имеют такой команды, но обладают тем фундаментальным набором команд, который обычно встречается на RISC- или CISC-компьютерах: `shift`, `add`, `and`, `load`, командами условного перехода и т. д. Предположим в целях иллюстрации, что компьютер имеет 32-битный размер слова, но большая часть рассматриваемых методов может быть легко адаптирована к другим размерам слова.

Подсчет заполнения решает две проблемы: вычисляет количество единичных битов в отдельном компьютерном слове и подсчитывает единичные биты в большом количестве слов, возможно, упорядоченных в массив. В каждом случае мы показываем, что очевидное, даже доведенное до совершенства решение, может быть превзойдено совершенно другими алгоритмами, для поиска которых требуется некоторая доля воображения. На первый план выходит применение стратегии «разделяй и властвуй», а на второй — применение определенной

логической схемы, которая привычна разработчикам компьютерной логики, но не слишком знакома программистам.

Основные методы

В первом приближении программист может подсчитать единичные биты в слове x , как показано в следующем решении, реализованном на языке Си. Здесь x является беззнаковым целым числом, поэтому сдвиг вправо будет осуществлен для нулевого бита:

```
pop = 0;
for (i = 0; i < 32; i++) {
    if (x & 1) pop = pop + 1;
    x = x >> 1;
}
```

На обычном RISC-компьютере цикл может быть скомпилирован в семь команд, относящихся к условному переходу. (Один из условных переходов предназначен для управления циклом.) Эти семь команд выполняются 32 раза, но одна из них (как мы можем предположить) примерно в половине случаев обходится, таким образом компьютер выполняет около $32 \times 6,5 = 208$ команд. Наверное, программисту не нужно будет долго думать, чтобы понять, что этот код без особых усилий может быть улучшен. Прежде всего на многих компьютерах исходящий счет от 31 до 0 производится более эффективно, чем счет восходящий, поскольку при нем экономится команда *сравнения*. А почему бы не пойти дальше и вообще избавиться от счета?

Просто цикл нужно выполнять до тех пор, пока x не станет нулем. При этом если в x встретятся старшие нулевые биты, будет исключена часть шагов перебора. Следующая оптимизация будет заключаться в замене *if*-проверки простым прибавлением к счетчику крайнего справа бита x . Тогда получится следующий код:

```
pop = 0;
while (x) {
    pop = pop + (x & 1);
    x = x >> 1;
}
```

В цикле останется только четыре или пять RISC-команд, в зависимости от того, потребуется или нет *сравнение* x с нулем, и только один переход. (Мы предполагаем, что компилятор перестроит цикл, чтобы условный переход оказался внизу.) Таким образом, будет задействовано максимум от 128 до 160 команд. Максимальное количество будет задействовано в том случае, если x начинается с единичного бита, а если у x будет много лидирующих нулей, то количество команд будет значительно меньше.

Некоторые читатели могут вспомнить, что простое выражение $x \& (x-1)$ – это x с выключенными самим младшим единичным битом, или 0, если $x = 0$. Таким образом, чтобы подсчитать в x единичные биты, можно их сбрасывать по одному до тех пор, пока результат не станет нулевым, подсчитывая количество сбросов. В результате получится следующий код:

```
pop = 0;
while (x) {
    pop = pop + 1;
    x = x & (x - 1);
}
```

Как и для предыдущего кода, в цикле будет задействовано четыре или пять инструкций, но цикл будет запускаться столько раз, сколько единичных битов будет содержаться в x . Это безусловное улучшение.

Комплементарный подход, применимый в том случае, если ожидается большое количество единичных битов, заключается во включении крайнего правого нулевого с помощью выражения $x = x | (x+1)$ до тех пор, пока результат не будет состоять из всех единиц (-1). В переменной p ведется подсчет количества выполненных итераций, и возвращается $32 - p$. (В другом варианте к исходному числу x может быть образовано дополнение или p может быть присвоено исходное значение 32, и подсчет может вестись с убыванием.)

Первая программа в этой серии слишком незатейлива, но другие вполне могут радовать глаз своей эффективностью, продуманностью и разумной практичностью. Первую программу можно было бы заставить работать значительно быстрее, путем развертки цикла, но другие две программы таким же изменением если и могут быть улучшены, то очень незначительно.

Можно также воспользоваться табличным поиском, возможно, осуществляя за один шаг побайтовый перевод x в количество единичных битов в этом байте. Код получается довольно коротким и должен быть очень быстродействующим на многих машинах (примерно 17 команд на типовой RISC-машине, не имеющей индексной загрузки). В следующем коде, $\text{table}[i]$, — это количество единичных битов в i , для i в диапазоне от 0 до 255:

```
static char table[256] = {0, 1, 1, 2, 1, 2, 2, 3, ..., 8};
pop = table[x & 0xFF] + table[(x >> 8) & 0xFF] +
      table[(x >> 16) & 0xFF] + table[x >> 24];
```

«Разделяй и властвуй»

Другой интересный и полезный подход к подсчету заполнения слова единичными битами основывается на парадигме «разделяй и властвуй». Алгоритм мог бы быть разработан исходя из следующих соображений: «Предположим, у меня есть способ подсчета заполнения 16-битного числа. Тогда я смогу его запустить для левой и правой половины 32-битного слова и сложить результаты, чтобы получить подсчет заполнения 32-битного слова». Если базовый алгоритм должен будет запускаться последовательно для двух половин, этот алгоритм не окупится, а на выполнение будет затрачено время, пропорциональное количеству анализируемых битов, поскольку тогда он займет $16k + 16k = 32k$ единиц времени, где k — константа пропорциональности, плюс еще время на выполнение команды сложения. Но если мы каким-то образом сможем проводить операции над двумя половинами слова параллельно, то показатель улучшится от, практически, $32k$ до $16k + 1$.

Для эффективного подсчета заполнения двух 16-битных чисел нам нужен способ производства такого подсчета для 8-битных чисел и проведения в параллель четырех таких вычислений. Если продолжить это рассуждение, то нам нужен способ подсчета заполнения 2-битных чисел и проведения в параллель 16 таких вычислений.

Алгоритм, о котором будет рассказано, никоим образом не зависит от запуска операций на отдельно взятых процессорах или на каких-нибудь необычных командах, таких как SIMD-команды¹, которые можно найти на некоторых компьютерах. Он использует лишь возможности, которые обычно можно найти на обычном однопроцессорном RISC- или CISC-компьютере.

Схема показана на рис. 10.1.

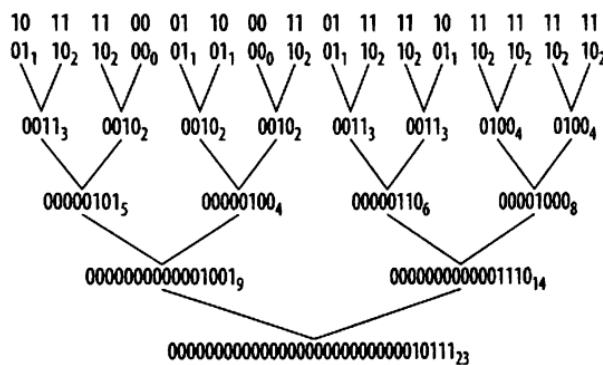


Рис. 10.1. Вычисление единичных битов с использованием стратегии «разделяй и властвуй»

Первая строка на рис. 10.1 представляет собой слово x , для которого мы хотим осуществить подсчет единичных битов. Каждое 2-битное поле во второй строке содержит количество единичных битов в том 2-битном поле, которое расположено непосредственно над ним. В подстрочном индексе находятся десятичные значения этих 2-битных полей. Каждое 4-битное поле в третьей строке содержит сумму чисел двух смежных 2-битных полей второй строки, с подстрочниками, показывающими десятичные значения, и т. д. Последняя строка содержит количество единичных битов в числе x . Алгоритм выполняется в $\log_2(32) = 5$ шагов, где каждый шаг содержит несколько команд сдвига и маскирования для сложения смежных полей.

Метод, проиллюстрированный на рис. 10.1, может быть реализован на Си следующим образом:

```
x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
```

¹ Команды с одним потоком команд и многими потоками данных оперируют параллельно с несколькими полями (такими как байты или полуслова) компьютерного слова. Например, 8-битная SIMD-команда add может складывать соответствующие байты двух слов без распространения переноса из одного байта к следующему.

```
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF);
```

(Константы, начинающиеся на Си с 0х, имеют шестнадцатеричное значение.) В первой строке использовано выражение $(x >> 1) \& 0x55555555$ вместо, возможно, более естественного $(x \& 0xAAAAAAA) >> 1$, поскольку показанный код свободен от генерации двух больших констант в регистре. Если в машине отсутствует команда and not, то экономится одна команда. То же самое замечание можно сделать и для других строк.

Понятно, что последний оператор И не нужен, поскольку результат $x >> 16$ должен начинаться с 16 нулевых битов, поэтому И не приводит к изменению значения $x >> 16$. Другие команды И могут быть опущены в том случае, если отсутствует опасность, что сумма поля может быть перенесена в пределы смежного поля. Также есть способ переделать код первой строки, уменьшив его на одну команду. Это ведет к упрощению, показанному в примере 10.1, выполняемому в 21 команду и свободному от переходов и ссылок в памяти.

Пример 10.1. Вычисление единичных битов в слове

```
int pop(unsigned x) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x + (x >> 4)) & 0x0F0F0F0F;
    x = x + (x >> 8);
    x = x + (x >> 16);
    return x & 0x0000003F;
}
```

Первое присваивание x основывается на первых двух элементах формулы:

$$\text{pop}(x) = x - \left\lfloor \frac{x}{2} \right\rfloor - \left\lfloor \frac{x}{4} \right\rfloor - \left\lfloor \frac{x}{8} \right\rfloor - \dots - \left\lfloor \frac{x}{2^{31}} \right\rfloor.$$

Здесь x должно быть большим или равным нулю. Если считать x беззнаковым целым числом, это уравнение может быть реализовано последовательностью из 31 *правого сдвига на один регистр* и 31 *вычитания*. В процедуре примера 10.1 используются первые два элемента этой последовательности над каждым 2-битным полем параллельно. Доказательство этого уравнения я оставляю за читателями.

К сожалению, код примера 10.1 потерял большую часть упорядоченности и элегантности того кода, из которого он произошел. Вследствие этого была утрачена ясная перспектива расширения этого кода для 64-битной машины. Трудно отказаться от всех этих возможностей ради сохранения команд.

«Разделяй и властвуй» является весьма важной методикой, которая должна быть где-то сверху в арсенале трюков каждого программиста. То же самое можно сказать и для разработчиков компьютерной логики. Другими примерами использования этой методики можно назвать хорошо всем известную методику двоичного поиска, метод сортировки, известный как Quicksort, и метод инвертирования битов слова.

Другие методы

Статья 169 в меморандуме НАКМЕМ¹ описывает алгоритм подсчета количества единичных битов в слове x путем использования первых элементов формулы, показанной в предыдущем разделе, для каждого 3-битного поля x отдельно, чтобы произвести слово из 3-битных полей, каждое из которых содержит количество содержащихся в нем единичных битов. Затем происходит сложение смежных 3-битных полей в 6-битные поля суммы, а затем суммирует 6-битные поля, вычисляя значение слова по модулю 63. Хотя изначально он разрабатывался для машины с 36-битным словом, алгоритм легко адаптируется к 32-битному слову. Свидетельством этому служит пример на Си, показанный ниже (длинные константы приведены в восьмеричном формате):

```
int pop(unsigned x) {
    unsigned n;
    n = (x >> 1) & 033333333333;           // Подсчет битов
    x = x - n;                            // в каждом трехбитном
    n = (n >> 1) & 033333333333;           // поле.
    x = x - n;
    x = (x + (x >> 3)) & 030707070707; // Шестибитная сумма.
    return x%63;                          // Сложение шестибитных сумм.
}
```

В последней строке используется беззнаковая функция деления по модулю. (Если длина слова была бы кратна трем, она могла быть и знаковой, и беззнаковой.) Понятно, что функция деления по модулю суммирует 6-битные поля, когда вы считаете слово x целым числом, записанным по основанию 64. После деления целого числа по основанию b на b - 1 остаток, для b большего или равного трем, соответствует сумме единиц, и, конечно же, он меньше, чем b - 1. Поскольку сумма единиц в данном случае может быть меньше или равна 32, результат выражения mod(x, 63) должен быть равен сумме единиц, имеющихся в x, которая считается равной количеству единичных битов в исходном x.

Для этого алгоритма требуется всего лишь 10 команд на DEC PDP-10, поскольку у этой машины есть машинная команда, вычисляющая остаток с его вторым операндом, непосредственно ссылающимся на полное слово в памяти. На стандартной RISC-системе потребуется 15 команд, если предположить, что машина предоставляет *беззнаковое деление по модулю* в виде одной команды (но не дает непосредственной ссылки на ближайшее полное слово или на находящийся в памяти операнд). Но, по-видимому, это не слишком быстродействующая команда, поскольку деление почти всегда является довольно медленной операцией. Также она не применима к 64-битному слову путем простого расширения констант, поскольку не работает со словами, чья длина превышает 62 бита.

¹ Майкл Билер (Michael Beeler), Р. Уильям Госпер (R. William Gosper) и Ричард Шреппель (Richard Schroeppel), «НАКМЕМ, MIT Artificial Intelligence Laboratory AIM 239, February 1972». Благодаря Генри Бакеру (Henry Baker) теперь этот меморандум доступен по адресу: <http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html>

Еще один весьма любопытный алгоритм предусматривает 31-кратный циклический сдвиг x влево, с суммированием 32 элементов¹. Отрицательное значение суммы и является $\text{pop}(x)$! То есть:

$$\text{pop}(x) = -\sum_{i=0}^{31} (x \ll i),$$

где суммирование вычислено по модулю размера слова, и конечная сумма интерпретируется как целое число двоичного дополнения. Это всего лишь новинка, которой на большинстве машин вряд ли можно будет воспользоваться, поскольку цикл выполняется 31 раз, что потребует 63 команды плюс к этому контроль над выходом цикла за установленные пределы. Разгадку секрета работоспособности этого алгоритма я оставляю за читателями.

Сумма и разница подсчета заполнения двух слов

При вычислении $\text{pop}(x) + \text{pop}(y)$ (если компьютер не имеет команды для подсчета заполнения) можно сэкономить время за счет использования первых двух исполняемых строк из примера 10.1 отдельно для x и y , а затем сложения x и y и исполнения над суммой последних трех частей алгоритма. После того как будут выполнены первые две строки из примера 10.1, x и y будут состоять из восьми 4-битных полей, максимальное значение каждого из которых не будет превышать четырех. Таким образом, их можно складывать без всякой опаски, поскольку максимальное значение в каждом из 4-битных полей суммы не будет превышать 8, и переполнения не возникнет. (В действительности таким способом могут быть объединены три слова.)

Эта же идея может применяться и при вычитании. При вычислении $\text{pop}(x) - \text{pop}(y)$ можно воспользоваться следующим выражением²:

$$\begin{aligned} \text{pop}(x)\text{pop}(y) &= \text{pop}(x) - (32 - \text{pop}(\bar{y})) = \\ &= \text{pop}(x) + \text{pop}(y) - 32 \end{aligned}$$

Затем используется методика, которая только что была описана для вычисления $\text{pop}(x) + \text{pop}(y)$. Код показан в примере 10.2. В нем задействовано 32 команды, а не 43, как для двух применений кода примера 10.1 с последующим вычитанием.

Пример 10.2. Computing $\text{pop}(x) - \text{pop}(y)$

```
int popDiff(unsigned x, unsigned y) {
    x = x - ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    y = ~y;
```

¹ Mike Morton, «Quibbles & Bits», Computer Language, Vol. 7, № 12, Dec. 1990, pp. 45–55.

² y с надчеркиванием означает единичное дополнение y , которое на Си записывается как $\sim y$.

```

y = y - ((y >> 1) & 0x55555555);
y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
x = x + y;
x = (x & 0xF0F0F0F0) + ((x >> 4) & 0x0F0F0F0F);
x = x + (x >> 8);
x = x + (x >> 16);
return (x & 0x0000007F) - 32;
}

```

Сравнение подсчетов заполнений двух слов

Иногда, не прибегая к подсчету, нужно узнать, какое из двух слов имеет большее единичное заполнение. Возможно ли такое определение без подсчета заполнения этих двух слов? Одним из способов может быть вычисление разности двух подсчетов заполнений, как в примере 10.2, и сравнение ее с нулем, но если ожидаемое единичное заполнение обоих слов довольно низкое или если существует сильная корреляция между отдельными битами, установленными в двух этих словах, есть другой, более предпочтительный способ.

Идея состоит в сбросе одиночного бита каждого слова до тех пор, пока одно из слов не превратится в нуль; тогда другое слово будет иметь большее единичное заполнение. Процесс происходит быстрее, чем в самом худшем или среднем варианте, если сначала сбрасываются единичные биты в одинаковых позициях каждого слова. Код показан в примере 10.3. Процедура возвращает отрицательное число, если $\text{pop}(x) < \text{pop}(y)$, нуль, если $\text{pop}(x) = \text{pop}(y)$, и положительное число (1), если $\text{pop}(x) > \text{pop}(y)$.

Пример 10.3. Сравнение $\text{pop}(x)$ и $\text{pop}(y)$

```

int popCmpr(unsigned xp, unsigned yp) {
    unsigned x, y;
    x = xp & ~yp;           // Сброс битов там,
    y = yp & ~xp;           // где оба бита равны 1.
    while (1) {
        if (x == 0) return y | -y;
        if (y == 0) return 1;
        x = x & (x - 1); // Сброс одного бита
        y = y & (y - 1); // каждого слова.
    }
}

```

После сброса общих единичных битов в каждом 32-битном слове максимально возможное количество единичных битов в обоих словах равно 32. Поэтому слово, в котором единичных битов меньше, может иметь их не больше 16, и цикл в примере 10.3 выполняется максимум 16 раз, на что в самом неудачном случае потребуется 119 команд, выполняемых на обычных RISC-машинах ($16 \times 7 + 7$). Моделирование с использованием однородно распределенных случайных 32-битных чисел показало, что усредненное единичное заполнение меньшего числа после сброса общих единичных битов составляет примерно 6.186. Это дает при обработке случайных 32-битных чисел средний показатель времени выполнения, равный примерно 50 командам, что хуже, чем при использовании примера 10.2.

Чтобы эта процедура смогла опередить процедуру из примера 10.2, количество единичных битов в x или y после сброса общих единичных битов должно быть не более трех.

Подсчет единичных битов в массиве

Простейший способ подсчета количества единичных битов в массиве (векторе), состоящем из целых слов при отсутствии команды *подсчета заполнения*, состоит в использовании в отношении каждого присутствующего в массиве слова процедуры наподобие той, что была показана в примере 10.1, и простом сложении результатов. Мы называем это примитивным методом. Если проигнорировать управление циклом, генерацию констант и загрузку из массива, на него будет затрачено по 16 команд на каждое слово: 15 для кода из примера 10.1 плюс одна на сложение.

Мы предполагаем, что процедура подверглась внутреннему расширению, маски загружены за пределами цикла, а у машины имеется достаточное количество регистров для хранения чисел, используемых в вычислении.

Другой способ заключается в использовании двух первых исполняемых строк из примера 10.1 в отношении групп по три слова, находящихся в массиве, и сложения трех частных результатов. Поскольку максимальное значение каждого частного результата в каждом 4-битном поле не превышает четырех, сумма трех полей имеет максимальное значение, не превышающее 12 в каждом 4-битном поле, поэтому переполнение не возникает. Эта идея может быть применена к 8- и 16-битным полям.

Программирование и компиляция этого метода показывают, что он по общему количеству команд, выполняемых на обычной RISC-машине, дает около 20 % экономии. Большая часть экономии не состоялась из-за потребности в дополнительных служебных командах. Мы не будем останавливаться на этом методе, потому что для решения этой задачи есть более подходящие способы.

Пожалуй, лучший способ был изобретен приблизительно в 1996 году Робертом Харли (Robert Harley) и Дэвидом Сиэлом (David Seal)¹. Он основан на схеме, названной *сумматором с запоминанием переносов* (carry-save adder, CSA) или 3:2 компрессором. CSA представляет собой простую последовательность независимых полных сумматоров², которая часто используется в схемах двоичного умножителя.

¹ Дэвид Сиэл (David Seal), Телеконференция *comp.arch.arithmetic*, 13 мая 1997 года. Роберт Харли (Robert Harley) был первым известным мне человеком, применившим CSA для решения этой проблемы, а Дэвид Сиэл показал очень хороший способ его использования для подсчета битов в больших массивах (показанный на рис. 10.2 и в примере 10.5), а также в массиве из 7 элементов (подобно схеме на рис. 10.3).

² Полный сумматор — это схема с тремя однобитными входами (битами, подлежащими суммированию) и двумя битами на выходе (суммой и переносом). См. Джон Хеннеси (John L. Hennessy) и Дэвид Паттерсон (David A. Patterson), «Computer Architecture: A Quantitative Approach», Morgan Kaufmann, 1990.

В системе записи булевой алгебры (соединение символов обозначает *И*, знак + обозначает *ИЛИ*, а знак \oplus обозначает *исключающее ИЛИ*) логика для каждого полного сумматора имеет следующий вид:

$$\begin{aligned} h &< ab + ac + bc = ab + (a + b)c = ab + (a \oplus b)c \\ 1 &< (a \oplus b) \oplus c \end{aligned}$$

где a , b и c — это однобитные входные элементы, 1 — это младший выходной бит (сумма), а h — это старший выходной бит (перенос). Замена $a + b$ в первой строке на $a \oplus b$ вполне равносильна, поскольку когда a и b имеют единичное значение, элемент ab превращает значение всего выражения в единицу. Если сначала присвоить $a \oplus b$ промежуточному значению, логика полного суммирования может быть выражена в пяти логических командах, каждая из которых параллельно ведет обработку 32 бит (на 32-битной машине). Мы будем ссылаться на эти пять команд как на $CSA(h, 1, a, b, c)$. Это «макрос», на выходе которого получаются h и 1 .

Один из способов применения операции CSA заключается в обработке элементов массива A группами по три элемента, сводя каждую группу из трех слов к двум, и применяя подсчет заполнения к этим двум словам. Эти два подсчета заполнения суммируются в цикле. После выполнения цикла итоговый подсчет заполнения массива представляет собой удвоенный просуммированный подсчет заполнения старшего бита CSA плюс просуммированный подсчет заполнения младшего выходного бита.

Следующая последовательность иллюстрирует процесс для 16-битного слова:

$$\begin{aligned} a &= 0110\ 1001\ 1110\ 0101\ 9 \\ b &= 1000\ 1000\ 0100\ 0111\ 6 \\ c &= 1100\ 1010\ 0011\ 0101\ 8 \\ \hline 1 &= 0010\ 1011\ 1001\ 0111\ 9 \\ h &= 1100\ 1000\ 0110\ 0101\ 7*2 = 14 \end{aligned}$$

Заметьте, что пара $(h, 1)$ каждого столбца, записанная в этом порядке, является двухбитным двоичным числом, значение которого равно количеству единичных битов в столбце, составленном из a , b и c . Таким образом, каждый единичный бит в h представляет два единичных бита в a , b и c , а каждый единичный бит в 1 представляет один единичный бит в a , b и c . Поэтому итоговое заполнение (показанное справа) — это удвоенное число единичных битов в h плюс число единичных битов в 1 , которые в итоге на иллюстрации дают число 23. Пусть n_c будет количеством команд, необходимых для выполнения CSA-операций, а n_p будет количеством команд, необходимых для подсчета заполнения одного числа. На типовой RISC-машине $n_c = 5$, а $n_p = 15$. Если проигнорировать загрузку данных из массива и управление циклом (код для которых от машины к машине изменяется незначительно), то ранее рассмотренный цикл займет $(n_c + 2n_p + 2) / 3 = 12.33$ команды на каждое слово массива ($\ll 2$ — это для двух сложений в цикле). Это число отличается от 16 команд на слово, необходимое для осуществления примитивного метода.

Существует и другой способ использования CSA-операции в более эффективной и в немного более компактной программе, которая приведена в примере 10.4.

В ней затрачивается $(n_c + n_p + 1) / 2 = 10.5$ команды на слово (игнорируя управление циклом и загрузки).

Пример 10.4. Подсчет заполнения массива, при котором элементы обрабатываются группами по два

```
#define CSA(h,l,a,b,c) \
{unsigned u = a ^ b; unsigned v = c; \
h = (a & b) | (u & v); l = u ^ v; } \
int popArray(unsigned A[], int n) { \
    int tot, i; \
    unsigned ones, twos; \
    tot = 0; // Инициализация. \
    ones = 0; \
    for (i = 0; i <= n - 2; i = i + 2) { \
        CSA(twos, ones, ones, A[i], A[i+1]) \
        tot = tot + pop(twos); \
    } \
    tot = 2*tot + pop(ones); \
    if (n & 1) // Последний, если он есть. \
        tot = tot + pop(A[i]); // тоже нужно прибавить. \
    return tot; \
}
```

Когда пример 10.4 откомпилирован, CSA-операции расширяются в следующий код:

```
u = ones ^ A[i]; \
v = A[i+1]; \
twos = (ones & A[i]) | (u & v); \
ones = u ^ v;
```

Чтобы опустить последующие загрузки уже загруженных величин, код полагается на компилятор, этот процесс известен как *унификация (com-tomining)*.

Существуют способы использования CSA-операции, позволяющие проводить дальнейшее сокращение команд, необходимых для подсчета заполнения массива. Проще всего с ними разобраться с помощью схематической диаграммы. Например, рис. 10.2 иллюстрирует способ программирования цикла, который берет сразу восемь элементов массива и сжимает их в четыре значения, помеченные как eights (восьмерки), fours (четверки), twos (двойки) и ones (единицы). При следующем прохождении цикла fours, twos и ones возвращаются обратно в CSA, а единичные биты в eights подсчитываются функцией подсчета заполнений, предназначеннной для слова, и этот подсчет аккумулируется. Когда будет обработан весь массив, итоговое заполнение будет вычислено следующим образом:

$$8 \times \text{pop}(\text{eights}) + 4 \times \text{pop}(\text{fours}) + 2 \times \text{pop}(\text{twos}) + \text{pop}(\text{ones})$$

В коде, показанном в примере 10.5, используется CSA-макрос, определенный в примере 10.4. Нумерация CSA-блоков на рис. 10.2 соответствует порядку CSA-макросов, вызываемых в примере 10.5. Время выполнения цикла, исключая загрузки массива и управление циклом, составляет $(7n_c + n_p + 1) / 8 = 6.375$ команд на каждое слово массива.

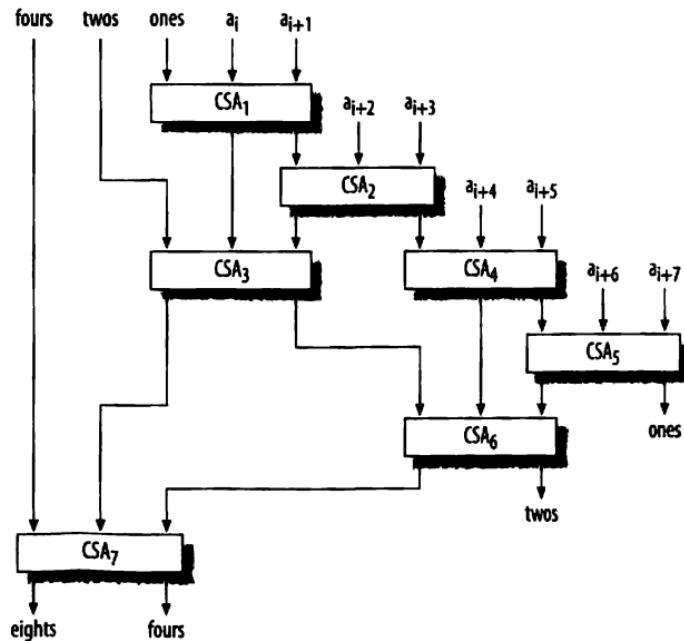
Пример 10.5. Подсчет заполнения массива, при котором элементы обрабатываются группами по восемь

```

int popArray(unsigned A[], int n) {
    int tot, i;
    unsigned ones, twos, twosA, twosB,
        fours, foursA, foursB, eights;
    tot = 0;                                // Инициализация.
    fours = twos = ones = 0;
    for (i = 0; i <= n - 8; i += 8) {
        CSA(twosA, ones, ones, A[i], A[i+1])
        CSA(twosB, ones, ones, A[i+2], A[i+3])
        CSA(foursA, twos, twos, twosA, twosB)
        CSA(twosA, ones, ones, A[i+4], A[i+5])
        CSA(twosB, ones, ones, A[i+6], A[i+7])
        CSA(foursB, twos, twos, twosA, twosB)
        CSA(eights, fours, fours, foursA, foursB)
        tot = tot + pop(eights);
    }
    tot = 8*tot + 4*pop(fours) + 2*pop(twos) + pop(ones);
    for (i = i; i < n; i++)                  // Простое сложение
        tot = tot + pop(A[i]);               // последних от 0 до 7 элементов.
    return tot;
}

```

Кроме схемы, показанной на рис. 10.2, операции CSA могут быть включены во многие другие структурные схемы.



К примеру, усиление распараллеливания на уровне команд могло бы быть получено за счет предоставления первых трех элементов массива одной операции CSA, а следующих трех элементов — второй операции CSA, что позволит командам этих двух CSA выполнять параллельно. Для увеличения параллелизма можно было бы также переставить местами три входных операнда CSA-макроса. По схеме, изображенной на рис. 10.2, легко можно понять, как использовать только первые три CSA-макроса, чтобы создать программу, которая обрабатывает элементы массива в группах по четыре, а также как ее можно развернуть для создания программ, обрабатывающих элементы массива в группах по 16 и более. На схеме также в какой-то степени отображается распределение загрузок, которое может пригодиться для машин с относительно низким пределом по этому параметру, который в любой момент может быть превышен.

В табл. 10.1 сведено количество команд, выполняемых при обобщенной реализации схемы, приведенной на рис. 10.2 для различных размеров групп. В значениях, приведенных в двух средних столбцах, проигнорированы команды на управление циклом и загрузки. В третьем столбце представлено общее количество команд цикла, приходящихся на одно слово входного массива, произведенное компилятором для типовой RISC-машины, не имеющей индексированной загрузки.

Таблица 10.1. Количество команд на слово при подсчете заполнения массива

Программа	Команды, за исключением загрузки и управления циклом		Количество команд в цикле (на выходе компилятора)
	Формула	Для $n_c = 5, n_p = 15$	
Примитивный метод	$np + 1$	16	21
Группы по 2	$(n_c + n_c + 1)/2$	10,5	14
Группы по 4	$(3n_c + n_c + 1)/4$	7,75	10
Группы по 8	$(7n_c + n_c + 1)/8$	6,38	8
Группы по 16	$(15n_c + n_c + 1)/16$	5,69	7
Группы по 32	$(31n_c + n_c + 1)/32$	5,34	6,5
Группы по 2^n			

Сокращение в конечном итоге количества команд для подсчета заполнения n слов с $16n$ — для примитивного метода, до $5n$ — для метода CSA, где 5 — это количество команд, необходимых для одной CSA-операции, стало приятным сюрпризом.

Для небольших массивов есть схемы лучше той, что показана на рис. 10.2. Например, для массива из семи слов очень эффективна схема, показанная на рис. 10.3¹. Для ее осуществления требуется $4n_c + 3n_p + 4 = 69$ команд, или по 9,86 команды на слово. Подобные схемы можно применить и к массивам размером $2k - 1$ слов, где k — любое положительное целое число. Схема для 15 слов выполняется за $11n_c + 4n_p + 6 = 121$ команду, или по 8,07 команды на слово.

¹ Сиэл (Seal), см. предыдущую ссылку.

Применение

Команда *подсчета заполнения* находит множество разнообразных применений. В начале главы уже упоминалось, что она используется для вычисления размера последовательности, представленной строкой битов. В этом представлении есть «универсум» с последовательно пронумерованными элементами. Последовательность представлена строкой битов, в которой бит i равен 1 только в том случае, если i является элементом последовательности.

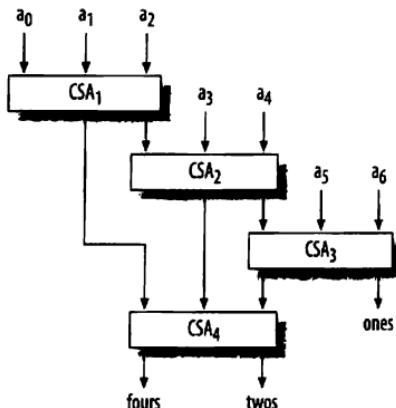


Рис. 10.3. Схема для подсчета итогового заполнения семи слов

Другой простой пример использования — вычисление расстояния Хемминга между двухбитными векторами, которое относится к теории кодов корректировки ошибок. *Расстояние Хемминга* — это просто количество различий между двумя векторами, которое определяется как¹:

$$\text{dist}(x, y) = \text{pop}(x \oplus y)$$

Команда *подсчета заполнения* может быть использована для вычисления в слове количества замыкающих нулей, для которого применяются следующие отношения:

$$\text{ntz}(x) = \text{pop}(Kx \& (x - 1)) = 32 - \text{pop}(x | -x)$$

(Читатели, не знакомые с этой смесью арифметических и логических операций, могут задержаться на несколько минут, чтобы выяснить, почему они работают.) Для функции $\text{ntz}(x)$ также находится множество разнообразных применений. К примеру, на некоторых первых компьютерах при обработке прерывания в специальном регистре должен был храниться бит «причины прерывания». Биты выставлялись в позиции, по которой устанавливался тип возникшего прерывания. Позиции были выбраны в порядке приоритетности, обычно биты прерываний с более высоким приоритетом занимали самые младшие позиции.

¹ Посмотрите, к примеру, главу по кодам коррекции ошибок в книге A. K. Dewdney, «The Turing Omnibus». Computer Science Press, 1989.

Одновременно могло быть установлено два или более битов. Для определения, какое из прерываний обрабатывать, диспетчерская программа должна была над значением специального регистра выполнять функцию ntz.

В другом примере использования *подсчета заполнения* предоставлялась возможность достаточно быстро получить прямой индексированный доступ к умеренно разреженному массиву A, представленному определенным компактным способом. В компактном представлении хранятся только определенные или не-нулевые элементы массива. Существует дополнительная строка битов bits, в которой имеется единичный бит для каждой битовой позиции i, для которой A[i] имеет определенное значение.

Поскольку обычно строка bits получается довольно длинной, она разбивается на 32-битные слова, где первый бит длинной строки становится нулевым битом (самым младшим битом) первого слова строки bits.

В качестве ускорителя предусмотрен также массив слов bitsum, в котором bitsum[j] — это общее число единичных битов во всех словах строки bits, которые предшествуют введенному значению j. Для массива, в котором определены элементы 0, 2, 32, 47, 48 и 95, это проиллюстрировано в следующей таблице:

bits	bitsum	data
0x00000005	0	A[0]
0x00018001	2	A[2]
0x80000000	5	A[32] A[47] A[48] A[95]

Главная задача заключается в следующем: заданный «логический» индекс i внутри заполненного массива перевести в «физический» разреженный индекс sparse_i, под которым элемент сохранен, если этот элемент существует, или получить какое-то свидетельство о том, что он не существует. Для массива в приведенной выше таблице мы хотим перевести 47 в 3, 48 в 4, а 49 — в сообщение «не существует». Для заданного логического индекса i соответствующий индекс sparse_i массива data задается в виде количества единичных битов в массиве bits, предшествующих биту, соответствующему i. Он может быть вычислен следующим образом:

```
j = i >> 5;           // j = i/32.
k = i & 31;            // k = rem(i, 32);
mask = 1 << k;        // "1" в позиции k.
if ((bits[j] & mask) == 0) goto no_such_element;
mask = mask - 1;       // единицы справа от k.
sparse_i = bitsum[j] + pop(bits[j] & mask);
```

Пространство, затрачиваемое на такое представление, составляет по два бита на каждую позицию в заполненном массиве.

Функция подсчета заполнения может быть использована для генерации биноминально распределенной случайной целочисленной величины. Чтобы сгенерировать целое число, выбранное из совокупности, предоставленной Binomial(t, p),

где t — это количество проб, а $p = 1/2$, сгенерируйте t случайных бит и подсчитайте в t битах количество единичных битов. Это может быть распространено на возможные значения p , отличные от $1/2^1$.

Если верить компьютерным слухам, то подсчет заполнения очень важен для Агентства национальной безопасности (National Security Agency). Похоже только, что никто (за пределами АНБ) не знает, для чего они его там используют, возможно, в криптографических задачах или при поиске в громадных количествах материала.

¹ См. раздел 3.4.1, проблема 27 в книге Donald E. Knuth, «The Art of Computer Programming: Seminumerical Algorithms», (Vol. 2, 3d ed.). Addison-Wesley, 1998.

11 Безопасная связь: технология свободы

Ashish Gulhati (Ashish Gulhati)

Я говорю лишь о компьютере, который должен прийти мне на смену. О компьютере, чьи самые простые рабочие параметры я не в состоянии даже вычислить — и все же я проектирую его для вас. Это компьютер, который сможет обработать Вопрос и дать Ответ на главный вопрос жизни, решающий все проблемы Вселенной, он настолько огромен и изощренно сложен, что вся органическая жизнь должна стать частью его оперативной матрицы.

Компьютер Deep Thought «Путеводитель для путешествующих автостопом по галактике»

В середине 1999 года я летел в Коста-Рику поработать с *Laissez Faire City*, группой, которая работала над созданием программной системы, помогающей проповедовать новую эру личной независимости¹.

Прежде всего, группа в *LFC* занималась разработкой пакета программ, предназначенного для защиты и совершенствования прав личности в наш цифровой век, включая простую и безопасную электронную почту, посредническую службу проведения интернет-диспутов, сетевую фондовую биржу, частную торговлю активами и работу с банковской системой. Мой интерес ко многим подобным технологиям был вызван довольно давно рассылками «шифропанков» и книгой Брюса Шнайера (Bruce Schneier) «Applied Cryptography» (издательство Wiley), и я уже прорабатывал возможности реализации некоторых из этих систем.

Наиболее фундаментальными из них были системы предоставления практически всем желающим стойкой и удобной конфиденциальной связи.

Когда я переступил порог «временно представительства» *LFC*, раскинувшегося на окраине Сан-Хосе в Коста-Рике, его сотрудники были заняты разработкой прототипа безопасной системы электронной почты, которую они называли

¹ См. «Sovereign Individual: Mastering the Transition to the Information Age», James Dale Davidson, Sir William Rees Mogg, Free Press, 1999.

MailVault. Она запускалась на Mac OS 9, использовала в качестве своей базы данных FileMaker и была написана на Frontier. Наверное, вам вряд ли бы захотелось запускать столь ответственную службу связи на такой мешанине технологий, но ничего другого программисты на тот момент еще не сделали.

Неудивительно, что система очень часто попадала в аварийную ситуацию и была крайне ненадежной. Группа *LFC* стояла перед кризисом доверия инвесторов, поскольку выпуски программ неоднократно откладывались, а первая бета-версия их ведущего продукта, MailVault, была далека от совершенства.

Поэтому в свое свободное время, остающееся от сетевой работы по контракту и системного администрирования в *LFC*, я приступил с чистого листа к созданию новой системы безопасной почты.

Эта система, которая сейчас носит название Cryptonite, разрабатывалась и тестиировалась от случая к случаю, в промежутках между разработками других проектов.

Первый работоспособный прототип Cryptonite был лицензирован для *LFC* в качестве MailVault beta 2 и был открыт для тестирования в сентябре 1999 года. Это была первая OpenPGP-совместимая система электронной почты, доступная для открытого использования, которая почти сразу была поставлена на тестирование инвесторами *LFC* и бета-тестерами. С той поры, в результате общения с пользователями, сообществом открытого программного обеспечения и рынком, Cryptonite получил всестороннее развитие. Хотя сам продукт не был открытым, он привел к созданию многочисленных компонентов, которые я в процессе разработки решил выпустить с открытым исходным текстом.

С чего все начиналось

Разработка системы Cryptonite, ее маркетинг и поддержка связанных с нею служб велись мной единолично в течение многих лет (благодаря также сильной поддержке и многим неоценимым идеям моей жены Бархи) и были невероятно интересным и полезным путешествием, не только с точки зрения разработки, но и в предпринимательском смысле.

Перед тем как перейти к основным элементам системы, я решил затронуть некоторые моменты, которые врезались в мое сознание в ходе этого проекта.

- Мой друг Ришаб Гош (Rishab Ghosh) однажды язвительно заметил, что возможности работать из любой точки мира, предоставляемые Интернетом тем хакерам, которые имеют проводную связь, сильно преувеличены, а многие из тех, кто создал этот миф, живут в небольшом местечке в Калифорнии. Для запускаемого в разработку независимого проекта очень большое значение имеет принципиальная возможность его реализации из любого места, и что работа над ним когда угодно может быть отложена и возобновлена. Я колдовал над системой Cryptonite на протяжении многих лет, путешествуя по четырем континентам, и это, наверное, была первая высококачественная прикладная программа, большая часть которой была разработана в горах Гималаев. (Раньше я свободно пользовался термином «проводная связь». В действительности наша связь в Гималаях обеспечивалась применением

пяти беспроводных технологий: спутниковой интернет-связи VSAT, Wi-Fi, Bluetooth, GPRS и CDMA.)

- Разрабатывая проект в одиночку и в свободное от основной работы время, следует помнить старую хакерскую мудрость: «Десять минут поисков в библиотеке могут сэкономить шесть месяцев лабораторных изысканий». Максимальное использование существующих библиотек программного кода имеет огромное значение. Поэтому для разработки системы я выбрал Perl, весьма популярный и гибкий язык высокого уровня, с уже сложившейся, богатой библиотекой, доступными программными модулями и тем, что у любителей Perl первым проявлением хакерской ленинсти является информирование о каждом конструкторском решении.
- Основным вопросом, особенно для тех программ, которые рассчитаны на конечного пользователя, становится простота использования. Предоставление пользователю простого, доступного интерфейса становится главным в функционировании их кода. Соображения простоты и удобства использования при разработке приложений обеспечения безопасности конечного пользователя имеют даже более существенное значение и были по-настоящему ключевым фактором в повышении ценности разработки системы Cryptonite.
- Чтобы с первых же шагов вырисовывалась перспектива успеха, будет вполне разумно начать с создания рабочего прототипа и после реализации основных функциональных элементов использовать процесс превращения прототипа в промышленный образец для перехода к промышленному развертыванию. Такой порядок может оказать огромное содействие становлению основной конструкции и структуры, прежде чем кодом воспользуются сотни или (я надеюсь!) миллионы пользователей.
- Нужно всегда стремиться к тому, чтобы система была как можно проще. Не поддавайтесь увлечению новейшими сложными и хитроумными технологиями, пока приложение действительно не будет испытывать необходимость в их использовании.
- Современные процессоры обладают достаточным быстродействием, и время, затрачиваемое программистами на разработку, порой ценится больше, чем процессорное время, но скорость работы приложений все же еще не утратила своей актуальности. Пользователи всегда ждут от своих приложений растворимости. Для веб-приложений, которыми одновременно пользуется множество людей, затраты на оптимизацию быстродействия всегда окупятся сторицей.
- Программное приложение как некая живая субстанция требует постоянного внимания, обновления, усовершенствования, тестирования, исправления, настройки, рыночного продвижения и поддержки. Его успех и привлекательность в экономическом смысле напрямую зависят от гибкости кода, достаточной для эволюционного развития, соответствия запросам потребителей и возможности многократного следования этим требованиям на протяжении многих лет.
- Ваша личная заинтересованность будет во многом способствовать решению проблемы. Это позволит не только легко переключаться с роли пользователя

на роль разработчика, но и гарантирует сохранение интереса к проекту и пять лет спустя, поскольку создание и рыночное продвижение программного приложения являются в большинстве случаев весьма долговременным занятием.

Разработка Cryptonite во многом стимулировалась моим желанием создать инструментарий, помогающий людям во всем мире обрести реальную свободу. И хотя порой непросто было разрабатывать систему в одиночку, я пришел к заключению, что индивидуально разрабатываемый проект, ко всему прочему, придает коду определенное стилистическое и структурное единство, которое редко достигается, если в его разработке участвуют несколько программистов.

Разбор проблем безопасного обмена сообщениями

При всем величии самой идеи защиты прав человека (которые еще будут обсуждаться) путем предоставления возможностей безопасной связи, ее приемлемая реализация не такая уж простая задача, как это может показаться. Криптографические системы на основе открытого ключа в принципе могут содействовать случайно создаваемой безопасной связи, но их практическая реализация очень часто требует неоправданных усложнений и оторвана от реальной действительности в отношении того, кто и как будет пользоваться подобными системами.

Главной проблемой, требующей решения при практической реализации криптографической системы на основе открытых ключей, является установление подлинности ключа. Чтобы послать кому-то закодированное сообщение, вам нужно иметь его открытый ключ. Если вы обманным путем можете быть втянуты в использование неверного открытого ключа, то ваша конфиденциальность просто исчезнет.

Для решения проблемы установления подлинности ключа существует два совершенно разных подхода.

Традиционный подход, связанный с инфраструктурой открытого ключа, — Public Key Infrastructure (PKI), как правило, основан на стандарте ISO X.509 и зависит от системы надежного управления сертификатами — Certification Authorities (CA), разработанной независимыми организациями, и во многом принципиально не подходит для удовлетворения реальных потребностей пользователей случайно создаваемых сетей¹. Реализации PKI пользуются значительным успехом в более структурированных областях, таких как корпоративные виртуальные частные сети (VPN) и установление безопасности веб-сайтов, но не получили широкого применения в реальной неоднородной среде электронной почты.

¹ Недостатки традиционной PKI были вкратце изложены Роджером Кларком (Roger Clarke) на веб-странице <http://www.anu.edu.au/people/Roger.Clarke/II/PKIMisFit.html>.

Другой подход представлен наиболее популярным из используемых в настоящее время решений по безопасному обмену сообщениями на основе открытых ключей: системой PGP Фила Циммерманна (Phil Zimmermann) и ее потомками, формализованными в виде протокола IETF OpenPGP. Этот протокол сохраняет гибкость и, по сути, децентрализованный характер криптографии с открытым ключом, облегчая распределенную ключевую аутентификацию через «сети доверия» и исключая зависимость от централизованной, иерархической системы управления сертификатами (CA), как это делается при использовании PKI (в том числе и главным конкурентом OpenPGP – системой S/MIME). Неудивительно, что S/MIME, доступная практически в любой популярной клиентской программе электронной почты, обладает значительно меньшей популярностью у пользователей, чем OpenPGP, несмотря на существенный дефицит почтовых клиентов, обладающих полной поддержкой OpenPGP.

Но подход, использующий «сети доверия», который при построении цепочек доверия по сертификации и подтверждения открытых ключей полагается

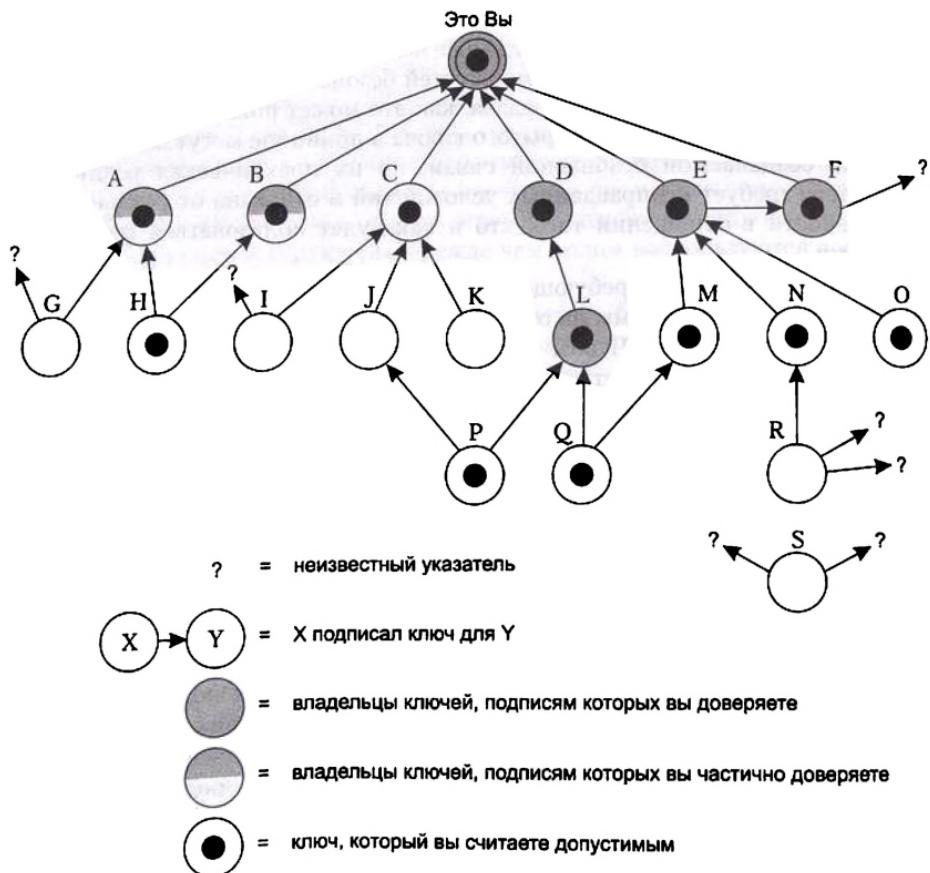


Рис. 11.1. Порядок признания ключей через сеть доверия

на пользователей, имеет свои собственные проблемы. И самая первая из них состоит во взаимосвязанных затруднениях как в получении гарантий, что пользователи понимают порядок использования сети доверия для подтверждения ключей, так и в необходимости достижения критической массы пользователей, чтобы любые два пользователя смогли найти при общении друг с другом надежный канал.

На рис. 11.1 показано, что при реализации сети доверия никакие трети лица произвольно не обозначаются как «вызывающие доверие». Каждый индивидуальный пользователь сам по себе является особо доверенным центром сертификации и может назначать изменяющиеся уровни доверия другим в целях проверки достоверности ключей. Ключ считается достоверным, если он сертифицирован непосредственно для вас кем-то другим, кому вы полностью доверяете сертифицировать ключи, или определяемым пользователем числом людей, каждый из которых обладает частичным вашим доверием по сертификации ключей.

Поскольку подход, связанный с использованием сети доверия, не использует попыток проведения сторонней аутентификации ключа, как это делается при подходе с использованием PKI, пользователи должны играть центральную роль в построении своих сетей доверия и установлении подлинности открытых ключей. В связи с этим при конструировании системы безопасного обмена сообщениями на основе OpenPGP на первый план выходят соображения удобства и простоты ее использования.

Ключевая роль удобства и простоты использования

Программы обеспечения конфиденциальности электронной почты зачастую требуют от пользователей перепрыгнуть массу барьеров, поэтому воспользоваться ими решаются далеко не все. Удобство пользования играет ключевую роль для любого решения системы безопасности, поскольку если система таким качеством не обладает, все закончится тем, что ее обойдут или будут использовать в незащищенном режиме, что в обоих случаях аннулирует ее предназначение.

Социологическое исследование удобства и простоты использования PGP, проведенное в университете Карнеги-Меллона в 1998 году, указало на специфические проблемы создания эффективного и удобного интерфейса для систем кодирования электронной почты и определило, что из 12 участников исследования, пользовавшихся электронной почтой, «только одна треть была в состоянии воспользоваться PGP, чтобы правильно подписать и зашифровать почтовое сообщение за отведенные 90 минут»¹.

Я считаю Cryptonite интересным проектом с точки зрения конструирования, безопасной, надежной и эффективной системой электронной почты с очень

¹ «Usability of Security: A Case Study», Альма Уиттен (Alma Whitten) и Дж. Д. Тайгер (J. D. Tygar), университет Карнеги-Меллона. <http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-155.pdf>.

высоким уровнем удобства использования. Я настроился на создание почтовой системы, в которой безопасность, построенная на основе OpenPGP, была бы встроена в саму структуру анализа электронной почты, чтобы помочь даже случайным пользователям эффективно применять OpenPGP для достижения конфиденциальности связи. Формат электронной почты был выбран именно потому, что он способен предоставить мощную технологию конфиденциальности связи всем, кто получает доступ через интернет-кафе или через мобильный телефон с веб-браузером, а не только тем, кто может запустить программное обеспечение безопасной электронной почты на мощном настольном компьютере.

Cryptonite был спроектирован, чтобы сделать кодирование информации стандартной составляющей ежедневной электронной почты не за счет скрытия сложности криптографических систем открытого ключа, на которых оно основано, а скорее, за счет придания элементам этих систем большей ясности и доступности для пользователя. Поэтому при проектировании Cryptonite основное внимание уделялось вопросам удобства пользования во многих его проявлениях.

Разработка функций пользовательского интерфейса на основе отзывов и исследования вопросов удобства и простоты использования

Социологические исследования, проведенные в университете Карнеги-Меллона, дали множество продуктивных идей для исходной конструкции, и многие ее возможности развивались на основе тестирования удобства использования самой программы Cryptonite обычными пользователями электронной почты. Интерфейс получился понятным, содержащим минимально необходимые и согласованные между собой элементы, до всех самых важных функций из любого места можно было добраться в основном за один-два клика. Важные представления о сути внутреннего устройства, извлеченные из результатов проведенного тестирования на удобство использования, включали необходимость сосредоточения основных элементов управления в клиентской почтовой программе, необходимость постоянного присутствия расшифрованных сообщений и, желательно, отображения структурной информации в представлении списка сообщений.

Окончательная трехпанельная компоновка, похожая на имеющуюся в программах электронной почты для настольных компьютеров, сложилась после тестирования однопанельного HTML-интерфейса и AJAX-интерфейса. В трехпанельном интерфейсе был воплощен оптимизированный опыт использования, не требующий перезагрузки страницы при каждом возвращении к списку сообщений, как при однопанельной конструкции, к тому же простой трехпанельный HTML-интерфейс легче поддавался переносу и был проще в реализации, чем AJAX, и к тому же не требовал значительного увеличения пропускной способности канала связи.

Полноценное и интуитивно понятное пользователю отображение объектов OpenPGP

Пользователю доступны все ключевые операции, включая генерацию, импорт и экспорт ключей; проверку подписей ключей и контрольных сумм; сертификацию ключей и лишение их сертификаций; а также публикацию

ключей на ключевом сервере и их извлечение оттуда. Все это дает пользователю полное управление над его собственной сетью доверия. Уровень достоверности и степень доверия к ключам отображаются в тексте в явном виде, а также в цветовых кодах перечня ключей. Значения доверия к ключам постоянно обновляются самыми последними данными связки ключей и данными базы доверия. Связка ключей (Key Ring) пользователяского интерфейса, отображенная на рис. 11.2, показывает уровень достоверности всех пользовательских отличительных признаков для каждого ключа, как в текстовом виде, так и в цветовой кодировке. Кроме этого, на ней значками показан тип ключа и для каждого ключа степень доверия к его владельцу (как в текстовом виде, так и в цветовой кодировке). Все подробности относительно любого ключа доступны по ссылке его редактирования – edit.

Предупреждения и реакция на безопасность пользовательских действий

Предоставление пользователям полномочий по управлению ключами сопряжено с риском, что они воспользуются своими возможностями таким образом, что ослабят безопасность системы. Поэтому на приложение возлагается также обучение пользователей безопасному совершению таких действий, как сертификация ключей, изменения степени доверия к ключам или подпись сообщений.

Все экраны Cryptonite, в которых допустимы действия, связанные с вопросами безопасности, содержат краткие, специально выделенные предупреждения о последствиях. И все они находятся на тех же экранах, а не в раздражающих пользователей всплывающих окнах.

Type	Identities	Owner Trust	Edit
Cert	Ashish Gulhati <ashish@neomailbox.net> (Ultimately Valid)	-	edit
Cert	Barkha Gulhati <barkha@neomailbox.com> (Ultimately Valid)	Ultimately trusted certifier	edit
Cert	Ashish Gulhati <ashish@neomailbox.com> (Ultimately Valid) Ashish Gulhati <hash@metropoli.org> (Ultimately Valid) Ashish Gulhati <agul@cpan.org> (Ultimately Valid)	Ultimately trusted certifier	edit
Cert	Vipul Ved Prakash, <mail@vipul.net> (Fully Valid)	Fully trusted certifier	edit
Cert	Adam Back <adam@cypherspace.org> (Fully Valid)	Fully trusted certifier	edit
Cert	Gordon Worley (Mac GPG) <macgpg@b3land.co> (Expired) Gordon Worley (Mac GPG) <redbird@b3land.co> (Expired)	Marginaly trusted certifier	edit
Cert	Ashish Gulhati <ashish@neomailbox.net> (Ultimately Valid)	Ultimately trusted certifier	edit
Cert	Philip R. Zimmermann <prz@pgp.com> (Fully Valid)	Fully trusted certifier	edit
Cert	Rishab Aiyer Ghosh <rishab@dxm.org> (Fully Valid)	Fully trusted certifier	edit
Cert	John Gilmore <gnu@cygnus.com> (Fully Valid) John Gilmore <gnu@taid.com> (Fully Valid) John Gilmore <gnu@freeswan.org> (Fully Valid)	Fully trusted certifier	edit

Рис. 11.2. На панели отображения связки ключей (Key Ring) предоставлена информация о ключах и степени доверия к ним

Встроенные ассоциации

Используемая в Cryptonite концепция идентификации пользователя построена на привязке к секретным ключам, находящимся на пользовательской связке. При отправке сообщения пользователи могут воспользоваться адресом From (От кого), который соответствует секретному ключу, находящемуся на их связке. Это помогает пользователю интуитивно и безусловно понять идею секретного ключа. Открытые ключи могут быть привязаны к контактам пользовательской адресной книги, что даст возможность их автоматического использования для автоматического кодирования информации там, где это доступно.

Наличие полноценного почтового клиента

Cryptonite прежде всего является почтовым клиентом, обладающим к тому же полной поддержкой системы безопасности на основе OpenPGP и встроенной системой управления ключами. При ее создании была поставлена важная задача достижения удобства пользования, при котором пользователю предоставляется полноценная почтовая клиентская программа, не позволяющая системе безопасности снизить уровень удобства пользования электронной почтой. Это требование выражается не только в предоставлении полного спектра свойств, ожидаемых пользователем в программе почтового клиента, но, что более важно, в предоставлении возможности пользователю вести поиск по своим почтовым папкам, включая поиск в тексте зашифрованных сообщений, без существенного усложнения по сравнению с обычными почтовыми клиентами, в которых все сообщения хранятся в незашифрованном виде.

Основы

Конечно, в наши дни прикладное программное обеспечение в значительной мере отдалено от «голого» оборудования и является надстройкой над многими уже существующими уровнями программного кода. Поэтому при запуске нового проекта правильный выбор основ должен стать ключевой отправной точкой.

По ряду причин я выбрал для создания Cryptonite язык Perl. Богатый набор многократно используемых модулей с открытым кодом, предоставляемый всеобъемлющей сетью архивов Perl – CPAN (<http://www.cpan.org>), помогает свести к минимуму потребности в написании нового кода там, где могут быть применены уже имеющиеся решения, а также позволяет извлечь немалые выгоды из гибкости интерфейсов и выбора параметров. Это подтверждалось как всем предыдущим опытом использования этого языка, так и опытом, приобретенным при разработке проекта Cryptonite.

Способность имеющегося в Perl XS API обращаться к библиотекам, написанным на Си и других языках программирования, предоставляла доступ к еще более широкому кругу библиотек. Важными преимуществами были также великолепная переносимость кода, написанного на Perl, и мощная поддержка объектно-ориентированного программирования. При разработке Cryptonite

предусматривалось, что он будет легко доступен модификации со стороны обладателей лицензий, а это условие тоже было легче выполнить, если писать эту программу на Perl.

Итак, система Cryptonite реализована на полностью объектно-ориентированном языке Perl. Разработка проекта привела к созданию ряда Perl-модулей с открытым кодом, к которым я открыл доступ через CPAN.

В качестве естественной платформы разработки была выбрана GNU/Linux, поскольку код, разработанный в Unix-подобной среде проще всего переносится на платформы развертывания, где он будет использоваться, в качестве которых могут фигурировать только другие Unix-подобные платформы. В то время никакая Windows или Mac система (тогда была выпущена только пробная версия OS X) не имела качеств, необходимых для запуска столь ответственного программного обеспечения, предназначенного для одновременного применения тысячами пользователей. Поскольку я предпочитал использовать в качестве операционной среды своего настольного компьютера именно Linux, то ничего менять не пришлось.

В 2001 году разработка и развертывание были перенесены на OpenBSD, а с 2003 года разработка продолжилась на OS X и OpenBSD (а также на Linux). OS X была выбрана за присущие этой операционной системе исключительное удобство переносимости основного рабочего стола в сочетании с ее Unix-подобной основой и возможностью запускать широкий спектр программных продуктов с открытым кодом. OpenBSD была выбрана в качестве платформы для развертывания за свою надежность, превосходную регистрацию проблем безопасности и направленность на качество кода и его проверку.

В качестве интегрированной среды разработки (IDE) использовалась Emacs, которая была выбрана за свою мощность, расширяемость и великолепную переносимость, включая карманные и переносные устройства, которые я часто использовал для разработки в пути. Я также по достоинству оценил возможности имеющегося в Emacs режима cperl, которому удается проводить довольно неплохое автоформатирование кода Perl, даже при том что «разобраться в синтаксисе Perl может только сам Perl».

Конструктивные цели и решения

Cryptonite задумывался как совместимая с OpenPGP система электронной почты, разработанная с учетом высокого уровня безопасности, масштабируемости, надежности и простоты использования. Другими важными целями разработки проекта были переносимость и расширяемость.

Начальным ключевым решением была разработка абсолютно независимого основного механизма, облегчающего применение разнообразных интерфейсов и кросс-платформенный доступ. Для специалистов по разработке интерфейсов было важно иметь возможность создавать интерфейсы, ничего не меняя в основном механизме. Абсолютное отделение ядра системы от интерфейса позволило бы провести эксперименты по применению разнообразных интерфейсов, которые затем можно было бы подвергнуть тестированию на удобство применения,

помогающее подобрать оптимальный интерфейс. Это отделение является также основной конструктивной особенностью, позволяющей в будущем создавать разнообразные интерфейсы, включая интерфейсы, разрабатываемые для портативных устройств типа сотовых телефонов и «карманных» компьютеров.

Эта конструкция предусмотрена для клиент-серверной системы, обладающей четко определенным внутренним API и ясным разделением функций и полномочий между механизмом Cryptonite и пользовательским интерфейсом. Поэтому интерфейсы к ядру могут быть выполнены на любом языке с применением любой UI-структуры. Для тестирования удобства пользования может быть разработан интерфейс, допускающий отправку отзывов.

Рассматривалась также возможность предоставления гибкости в развертывании системы за счет выбора выполнения криптографических операций как на сервере, так и на машине, принадлежащей пользователю. У обоих подходов имеются свои преимущества и недостатки.

Хотя в принципе было бы желательно ограничить криптографические операции машиной пользователя, эти машины на практике довольно часто физически ненадежны и подвержены воздействию шпионских программ. С другой стороны, сервер может иметь преимущество за счет высокой физической защищенности и целенаправленного программного обслуживания со стороны специалистов, делая криптографию на стороне сервера (особенно в сочетании с аппаратной эстафетной аутентификацией) более безопасным выбором для многих пользователей. Это явилось еще одной причиной выбора для реализации системы языка Perl, поскольку его высокая мобильность позволила бы при необходимости выполнить приложение (или его компоненты) как на сервере, так и на пользовательских машинах.

Объектно-ориентированная реализация должна была помочь сохранить легкость понимания, расширения, обслуживания и модификации кода на протяжении многих лет.

Поскольку для обладателей лицензии и конечных пользователей предусматривался доступ к исходной форме кода, отдельной важной задачей ставилась его хорошая читаемость и доступность.

Конструкция основной системы

Исходная конструкция Cryptonite представлена на рис. 11.3.

Большая часть работы делается классом Cryptonite::Mail::Service, который определяет высокоуровневый служебный объект, осуществляющий все основные функции системы Cryptonite. Методы этого класса просто выполняют операции на основе своих аргументов и возвращают код состояния и результаты операции, если таковые имеются. Все методы не являются интерактивными, и в этом классе отсутствует код интерфейса:

```
package Cryptonite::Mail::Service;
```

```
sub new { # Object constructor
```

```
}
```

```

sub newuser { # Create new user account.
}

sub newkey { # Generate a new key for a user.
}

```

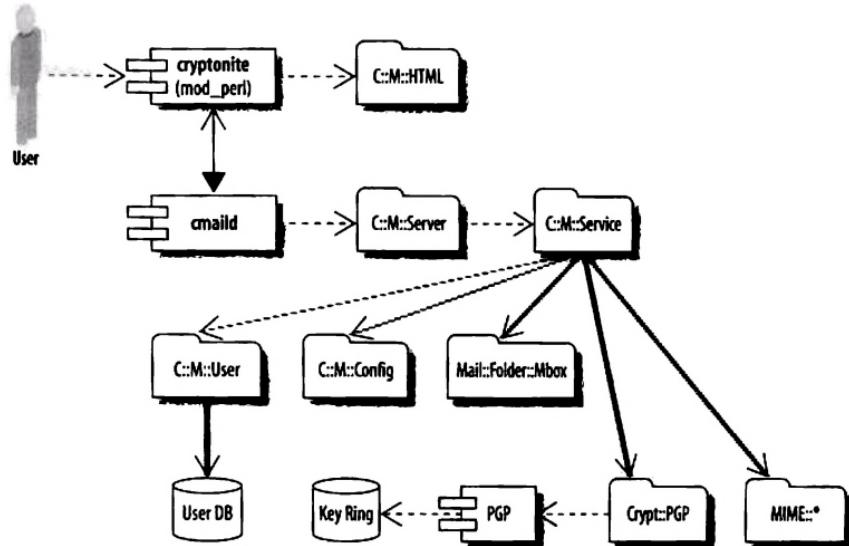


Рис. 11.3. Исходная конструкция Cryptonite (C::M является сокращением для Cryptonite::Mail)

В Cryptonite::Mail::Service инкапсулированы все основные функции системы, включая пользовательское создание и управление; создание, открытие и закрытие папок; отправку, удаление и копирование почты; шифровку, расшифровку и проверку подписи; а также анализ составных MIME-сообщений.

Класс Service используется в Cryptonite::Mail::Server для реализации сервера, который получает преобразованные в последовательную форму вызовы API Cryptonite, и отправки их объекту Service.

Преобразование в последовательную форму первоначально достигалось через вызовы SOAP, но разбор и обработка SOAP-объектов превносили слишком много ненужных усложнений и непроизводительных издержек. Поэтому взамен была реализована простая доморощенная схема сериализации. (По прошествии семи лет, если судить по материалам, опубликованным по адресу <http://wanderingbarque.com/nonintersecting/2006/11/15/the-s-stands-for-simple>, и имеющимся там отзывам, это была действительно удачная замена.) А вот как выглядит в Cryptonite::Mail::Server командный диспетчер:

```
package Cryptonite::Mail::Server;

use Net::Daemon;

use vars qw(@ISA);
use Cryptonite::Mail::Service;

@ISA = qw(Net::Daemon);

my $debug = 1;
my $cmail = new Cryptonite::Mail::Service;

sub process_request {
    my $self = shift; my ($retcode, $input);
    # Помещенный в eval перехват исключения по истечении срока.
    eval {
        local $SIG{ALRM} = sub { die "Timed Out!\n" };

        # истечение времени ожидания после 2 минут без ввода.
        my $timeout = 120;

        my $previous_alarm = alarm($timeout);
        while( <$STDIN> ){
            s/\r?\n//;

            # получение аргументов caller, command and cmd.
            my ($caller, $command, @args) = split /(?<!\\):/;
            $debug ? $debug == 2 ? warn "$$: $_\n" :
                warn "$$: $caller:$command:$args[0]\n" : '';

            # Недезактивированные разделители аргументов в потоке.
            for (@args) { s/(?<!:);(:)?/;/sg; s/:/:/sg }
            return if $command =~ /^$quit$/i;

            # Подтверждение достоверности команды.
            my $valid = $cmail->valid_cmd;
            if ($command =~/$valid/x) {
                # Вызов служебного метода Call.
                $ret = join ("\n", ($cmail->$command (@args), '')):
                print STDOUT $ret;
            }
            else {
                # Неверная команда.
                print STDOUT ($cmail->cluebat (ECOMMAND, $command) . "\n");
            }
            alarm($timeout);
        }
        alarm($previous_alarm);
    };
    if( $@ =~/timed out/i ){
        print STDOUT "Timed Out.\r\n";
        return;
    }
}
```

Cryptonite Mail Daemon (cmaild) получает сериализованный вызов метода через Unix или TCP сокеты, вызывает метод служебного объекта и возвращает результатирующий код (+OK или -ERR) вместе с воспринимаемым человеком сообщением (например «Все под контролем!») и необязательные возвращаемые значения (такие как перечень сообщений в папке или текст части сообщения). Если возвращается несколько строчек возвращаемого значения, сообщение о состоянии показывает, сколько строк для чтения должен ждать клиент.

Сервер создает ответвление нового процесса при каждом подключении нового клиента, поэтому встроенная в Perl функция тревоги `alarm` используется для отправки каждому новому служебному процессу определенного количества `SIGALRM $timeout` секунд, которое должно пройти после получения от клиента последнего сообщения, после чего произойдет истечение срока обслуживания и разрыв связи к клиентом.

Блок тестирования

Поскольку автоматизированное тестирование является ключевым компонентом долговременной разработки, я одновременно с кодом проекта создал блок тестирования.

Полное отделение ядра системы от интерфейса облегчило раздельное тестирование обоих компонентов, а также возможность проводить экспресс-диагностику ошибок и точно определять их место в программном коде. Создание тестов для `cmaild` свелось к вызову его методов с приемлемыми (и неприемлемыми) входными данными и слежению за тем, что они возвращают ожидаемые коды и значения.

Блок тестирования для `cmaild` использует вызов из клиентского API `cmdopen` (чтобы подключиться к Cryptonite Mail Daemon), `cmdsdel` (чтобы послать к daemon API-вызов) и `cmdayt` (чтобы послать серверу ping: «Ты здесь?»):

```
use strict;
use Test;

BEGIN { plan tests => 392, todo => [] }

use Cryptonite::Mail::HTML qw (&cmdopen &cmdsdel &cmdayt);

$Test::Harness::Verbose = 1;

my ($cmailclient, $select, $sessionkey);
my ($USER, $CMAILID, $PASSWORD) = 'test';
my $a = $Cryptonite::Mail::Config::CONFIG{ADMINPW};

ok(sub {                      # 1: cmdopen
    my $status;
    ($status, $cmailclient, $select) = cmdopen;
    return $status unless $cmailclient;
    1;
}. 1);

ok(sub {                      # 2: newuser
    my $status;
    $status = cmdsdel($USER, $CMAILID, $PASSWORD);
    return $status;
}. 1);

ok(sub {                      # 3: ping
    my $status;
    $status = cmdayt();
    return $status;
}. 1);
```

```

my $status = cmdsend('test.pl', $a, $mailclient, $select,
                     'newuser', $USER);
return $status unless $status =~ /^.+OK.*with password (.*)$/;
$PASSWORD = $1;
1;
}. 1);

```

Функциональный прототип

Для первого прототипа, чтобы создать на скорую руку основную пользовательскую базу данных, я воспользовался простым, перманентным объектным модулем `Persistence::Object::Simple` (который был написан для ранее разрабатывавшегося мною проекта моим другом Випулом (Vipul)). Использование перманентных объектов помогло сохранить чистоту и интуитивную понятность кода, а кроме того, предоставило прямой способ создания движков базы данных (для этого достаточно было создать или извлечь подходящий класс `Persistence::Object::*`).

В конце 2002 года Мэтт Сержант (Matt Sergeant) создал для Perl-разработчиков другой простой путь перехода от прототипа к конечному продукту, модуль `DBD::SQLite`, что-то вроде «автономной системы управления реляционной базой данных (RDBMS) в DBI-драйвере», которую можно было использовать в процессе разработки для быстрого создания прототипов кода баз данных, обходясь без полноценного движка баз данных. Тем не менее лично я предпочитаю элегантность и простоту перманентных объектов, чем засоренность своего кода SQL-запросами и DBI-вызовами.

Получаемая системой Cryptonite почта сохранялась в обычных mbox-файлах, вполне подходивших для работы прототипа. Разумеется, создание конечного продукта потребует использования более сложного хранилища электронной почты. В качестве окончного блока шифрования я решил воспользоваться самой PGP, чтобы избежать переписывания (и дальнейшей поддержки) всей группы функций шифрования, уже присутствующей в PGP.

Затем появилась программа GnuPG, и я взял ее на заметку для возможного использования в будущем при осуществлении криптографической поддержки. Итак, я написал `Crypt::PGP5`, чтобы инкапсулировать функциональность PGP5 в Perl-модуле. Этот модуль доступен в CPAN (хотя я его уже не обновлял целиком вечно).

Для криптографического ядра `Crypt::PGP5` я мог бы воспользоваться частной библиотекой PGPSDK, но не захотел связываться с созданием для нее Perl-интерфейса, на который потребовалось бы чуть ли не больше работы, чем при использовании двоичной версии PGP. Итак, с долей здоровой лени, присущей приверженцам языка Perl, и памятуя о принципе TMTOWTDI¹, я решил использовать для автоматического взаимодействия с двоичным кодом PGP

¹ «There's More Than One Way To Do It», то есть: «Чтобы что-то сделать, всегда есть несколько способов» – основной принцип образа жизни Perl-программистов.

модуль `Expect`, воспользовавшись тем же самым интерфейсом, который был доступен пользователям программы. Для работы первого прототипа этого было вполне достаточно.

Основной веб-интерфейс для заполнения HTML-шаблонов был разработан с использованием модуля `Text::Template`. Весь код, связанный с веб-интерфейсом, включая управление сессиями, содержался в модуле `Cryptonite::Mail::HTML`.

Система-прототип была готова уже спустя три месяца, частично затраченных на ее программирование. В ней был реализован полноценный веб-интерфейс, основы MIME-поддержки, OpenPGP-шифрование, расшифровка, подпись и проверка ее достоверности, регистрация нового пользователя в интерактивном режиме, а также новая и интересная альтернативная система ввода паролей для аутентификации: `PassFaces` из `ID Arts`.

Завершение, подключение, обкатка

После разработки исходного прототипа `Cryptonite` в Коста-Рике я продолжил независимую работу над этой системой. После столь необходимой подчистки кода (разработка прототипа носила лихорадочный характер, и на переработку кода и его тестирование времени не хватало) я работал над рядом модулей Perl и компонентами, которые пригодились бы в дальнейшем для перехода от простого прототипа к расширяемому продукту. Эта работа включала `Crypt::GPG` (с интерфейсом, который был практически идентичен интерфейсу `Crypt::PGP5`, поскольку для переключения криптографических операций `Cryptonite` на `GnuPG` требовалось несколько больше усилий, чем изменение одной строчки кода), а также `Persistence::Database::SQL` и `Persistence::Object::Postgres` (обеспечивающей объектную живучесть в базе данных `Postgres`, с интерфейсом, похожим на тот, что применялся к `Persistence::Object::Simple`, с помощью которого создавался практически беспроblemный внутренний переключатель базы данных).

В `Persistence::Object::Postgres`, как и в `Persistence::Object::Simple`, используется ссылка, созданная функцией `bless1` на хэш-контейнер, предназначенный для хранения пар ключ–значение, которые могут быть переданы базе данных с помощью вызова метода `commit`. В нем также используется имеющийся в Perl `Tie`-механизм, привязывающий большие объекты `Postgres` (BLOB) к описателям файлов, позволяя использовать естественный доступ на основе описателей файлов к большим объектам в базе данных.

Одним из главных преимуществ `Persistence::Database::SQL` над `Persistence::Object::Simple`, несомненно, является возможность создавать точные запросы к реальной базе данных. Например, при использовании `Persistence::Object::Simple` отсутствуют явные способы быстрого поиска конкретной пользовательской записи, тогда как в `Persistence::Database::SQL` получение определенной записи пользователя из базы данных труда не представляет:

¹ В Perl ссылка становится объектом, когда она связана с классом функцией `bless`, поэтому `bless`-ссылка – это всего лишь используемый в Perl термин объекта.

```

sub _getuser { # Get a user object from the database.
    my $self = shift; my $username = shift;
    $self->db->table('users'); $self->db->template($usertmpl);
    my ($user) = $self->db->select("WHERE USERNAME = '$username'");
    return $user;
}

```

Используя `Persistence::Object::Simple`, придется либо осуществлять перебор всех permanentных объектов в каталоге данных, либо заниматься нужной работой, обрабатывая с помощью grep-функции текстовые permanentные файлы в каталоге данных.

Во многих отношениях интерфейс `Persistence::Object::Postgres` очень похож на интерфейс `Persistence::Object::Simple`. Для модификации объекта из любого из этих модулей используется идентичный код:

```

my $user = $self->_getuser($username);
return $self->cluebat (EBADLOGIN) unless $user and $user->timestamp;
$user->set_level($level);
$user->commit;

```

Переключение с простой текстовой базы данных на реальную СУБД было реализовано после того, как основная часть кода прототипа работала в принципе без сбоев, и ознаменовало собой вторую стадию разработки Cryptonite: получение системы, готовой к реальному развертыванию. Для разработки прототипа `Persistence::Object::Simple` вполне подходил, поскольку не требовал наличия готового к развертыванию сервера базы данных, и объекты хранились в обычных текстовых файлах, которые легко просматривались при отладке.

Использование гомоморфных интерфейсов для `Crypt::GPG` и `Persistence::Object::Postgres` позволило реализовать эти весьма существенные изменения (внутренних блоков шифрования и базы данных) с минимальным объемом редактирования кода `Cryptonite::Mail::Service`.

Обновление хранилища электронной почты

Для хранения пользовательской почты в первом прототипе вполне подходили и простые твох-файлы, но для производственной системы нужно было иметь более эффективную возможность доступа и обновления индивидуальных сообщений, чем та, которую предоставлял единственный линейный файл почтового ящика. Мне также хотелось приблизиться к решению очень важной задачи обеспечения отказоустойчивости путем дублирования почтового хранилища.

Некоторые требования к хранилищу почты были продиктованы соображениями удобства пользования. В Cryptonite, в отличие от большинства программ-клиентов электронной почты, информация о MIME-структурах сообщений будет видна пользователям в списке сообщений. Это позволит пользователям прямо в списке сообщений увидеть, какие сообщения были зашифрованы и (или) подписаны. Наличие информации о составных частях сообщения к тому же дало бы возможность пользователю открывать подраздел сообщения напрямую. Части сообщения, как показано на рис. 11.4, отображены в виде значков в самом правом столбце страницы списка.

Date	From	Subject	Size (KB)	Parts
6:43 AM	Ashish Gubhatri	Photos...	17K	
6:38 AM	Ashish Gubhatri	No subject	2K	
6:15 AM	reddash	[The Sovereign Cyborg]...	1.2K	
9:27 AM	Angelina Rowe	Bait prices for u	11.6K	
12:46 AM	"Lara Sheldon"	Cursos completos de Tango	5.3K	
12:07 AM	"John Williams"	Do you know why Katherine the...	1.9K	

Рис. 11.4. Список сообщений с составными частями

Чтобы обеспечить подобную визуализацию, от хранилища электронной почты может потребоваться эффективное предоставление точной информации о MIME-структуре перечня сообщений. Дополнительная сложность создавалась тем, что спецификация OpenPGP/MIME позволяет вкладывать MIME-части вовнутрь подписанных и (или) зашифрованных частей, поэтому только хранилище электронной почты, подготовленное к работе с OpenPGP/MIME, может вернуть точную информацию о MIME-структуре зашифрованных или подписанных сообщений.

Поэтому я решил на основе модуля Mail::Folder создать внутреннее SQL-хранилище электронной почты, обладающее множеством возможностей IMAP4rev1-сервера. Ядром этой системы является класс Mail::Folder::SQL, основанный на Mail::Folder и использующий Persistence::Object::Postgres. Все это относится к тем временам, когда IMAP еще не получил столь широкой поддержки.

Я решил не использовать в качестве хранилища сообщений существующий IMAP-сервер, потому что предвидел необходимость в некоторых особенностях, которые большинство этих серверов в нужной мере не поддерживало, например такие как дублирование почтового хранилища и способность извлекать подробную информацию о структуре MIME-сообщения, без необходимости извлечения и разбора всего сообщения.

Даже если некоторые IMAP-серверы и могут отвечать моим потребностям, я также не хотел, чтобы Cryptonite был зависим и привязан к возможностям, предоставляемым конкретной реализацией того или иного IMAP-сервера. В конечном счете это решение оказалось правильным, даже при том, что оно потребовало приложения массы усилий по разработке кода, который затем утратил свою главенствующую роль в системе.

Дублирование хранилища электронной почты было разработано с использованием двух разработанных мною Perl-модулей: Replication::Recall и DBD::Recall, в которых используется созданная Эриком Ньютоном (Eric Newton) оболочка Recal (<http://www.fault-tolerant.org/recall>), предназначенная для дублирования баз данных на нескольких серверах. Замысел заключался в использовании ее в качестве прототипа и для построения в будущем по заказу новой системы дублирования базы данных.

С обновленной системой шифрования, базой данных и внутренним хранилищем электронной почты и с новой, подчищенной основой в октябре 2001 года в Интернете появилась первая внешняя бета-версия Cryptonite. Она была протестирована многими пользователями, обладающими различным уровнем компьютерного мастерства, некоторые из которых даже использовали ее в качестве

основного почтового клиента. Тестирование удобства пользования в рамках внешней бета-версии показало, что новички могли вполне успешно генерировать и импортировать ключи, а также посыпать и читать зашифрованные и подписанные сообщения.

Сохранность дешифрованной информации

Основная особенность шифрующего почтового клиента заключается в способности сохранять расшифрованные сообщения доступными в открытой форме на протяжении сеанса работы пользователя. Безопасный почтовый клиент, не обладающий этим свойством, может вызвать раздражение и стать неэффективным в использовании, поскольку требовал бы долгого набора ключевой фразы и ожидания расшифровки каждый раз, когда возникнет желание прочитать зашифрованное сообщение или что-нибудь в нем найти.

Сохранность в Cryptonite ранее расшифрованного сообщения была достигнута за счет создания нового класса `Mail::Folder`, основанного на `Mail::Folder::SQL`. `Mail::Folder::Shadow` передаст полномочия доступа к почтовому ящику *теневой папке*, если сообщение имеет в ней свою копию; в противном случае будет получен доступ к основной (или подвергаемой *теневому копированию*) папке.

С помощью этих средств расшифрованные сообщения могут храниться на время сессии в теневой папке, и для добавления функции хранения расшифрованной информации потребуется небольшая модификация кода, вместо включения модуля `Mail::Folder::Shadow` везде, где был использован `Mail::Folder::SQL`.

Все волшебство `Mail::Folder::Shadow` осуществляется простой, настраиваемой таблицей передачи полномочий:

```
my %method = 
qw( get_message 1 get_mime_message 1 get_message_file 1 get_header 1
    get_mime_message 1 mime_type 1 get_mime_header 1 get_fields 1
    get_header_fields 1 refile 1 add_label 2 delete_label 2
    label_exists 2 list_labels 2 message_exists 1 delete_message 5
    sync 2 delete 2 open 2 set_header_fields 2 close 2 DESTROY 2
    get_mime_skeleton 1 get_body_part 1):;
```

`Mail::Folder::Shadow` делегирует вызовы метода в соответствии с предопределением теневой папке, основной папке, подвергаемой теневому копированию, или им обеим. Применение имеющегося в Perl мощного средства `AUTOLOAD`, предоставляющего механизм управления методами, не имеющими явного определения в классе, является простым способом выполнения этого делегирования, который также предоставит простой механизм, позволяющий провести тонкую настройку обработки методов.

Методы, предназначенные для проверки теневого хранилища, такие как `get_message` и `get_header`, делегированы теневому хранилищу, если считается, что сообщение присутствует в теневой папке; в противном случае они делегируються основной папке, подвергаемой теневому копированию. Другие методы, такие как `add_label` и `delete` (удаляющий папку), должны быть отправлены как теневой, так и основной папке, поскольку эти послания должны изменить состояние не только основной, но и теневой папки.

Тем не менее другие методы, такие как `delete_message`, могут получать доступ к списку сообщений через ссылку на массив. Некоторые сообщения из списка могут быть подвергнуты теневому копированию, а другие могут не иметь копии. Имеющееся в `Mail::Folder::Shadow` средство `AUTOLOAD` обрабатывает подобные методы путем построения двух списков из переданного ему списка сообщений, один из которых содержит сообщения, подвергшиеся теневому копированию, а другой – те сообщения, которые ему не подвергались. Затем оно вызывает метод для обеих папок, основной, подвергающейся теневому копированию, и теневой, содержащей скопированные сообщения, и только для папки, подвергающейся теневому копированию, для тех сообщений, которые не были скопированы.

На практике в результате всего этого `cmaild` может продолжать использование папок в прежнем порядке и прятать расшифрованные сообщения в теневой папке на время существования сессии.

Чтобы обеспечить работу этого механизма, в `Mail::Folder::Shadow` существует несколько дополнительных методов, включая `update_shadow`, который используется для сохранения расшифрованных сообщений в теневой папке; `delete_shadow`, который используется для удаления по запросу пользователя отдельного сообщения, подвергшегося теневому копированию; и `unshadow`, который используется для удаления всех сообщений в теневой папке до прерывания сессии.

`Mail::Folder::Shadow` позволяет предложить на время сессии постоянное существование расшифрованных сообщений и осуществить поиск внутри зашифрованных сообщений. Эти два важных с точки зрения пользователя свойства редко реализуются в существующих на данный момент поколениях OpenPGP-совместимых почтовых систем.

Гималайский хакинг

В течение 2000–2001 годов я мог работать над Gryptonite лишь время от времени, в силу как других обязательств, так и того, что разработка проекта требовала спокойствия и тишины, которых мне так недоставало во время путешествия и жизни в хаосе, какофонии и замусоренности индийских городов.

Летом 2002 года мы с женой взяли отпуск и поехали в Гималаи, где я наконец-то смог выкроить время для завершения написания главных участков программного кода, включая добавление важных свойств управления ключами к Grypt::GPG и создание интегрированного интерфейса для управления ключами, который был очень важной частью всего механизма сети доверия. Ядро этого управляющего интерфейса, диалоговое окно `Edit Key`, показано на рис. 11.5. Оно дает возможность проводить проверку контрольной суммы (`fingerprint`), просматривать и создавать сертификаты идентичности пользователя и присваивать ключам значения степени доверия.

Я также перенес систему на OpenBSD, которая должна была стать конечной платформой развертывания.

У нас уже были все остальные главные компоненты безопасного почтового сервиса, и поскольку еще требовалось некоторое время для подготовки Cryptonite к публичному использованию, мы решили не останавливаться и немедленно запустить коммерческий сервис безопасной электронной почты. Это дало мне возможность тратить больше времени на разработку Cryptonite и безотлагательно приступить к созданию сообщества тестировщиков.

Итак, в середине 2003 года мы запустили почтовую службу Neomailbox, обеспечивающую безопасность IMAP, POP3 и SMTP.

В последующие годы стало очевидно, что это был превосходный шаг, который помогал финансирувать разработку, освобождая меня от потребности брать другую работу по контракту, и одновременно обеспечивал тесный контакт с рынком безопасного, частного обмена сообщениями.

Осенью 2003 года мы создали почти что постоянную разработочную базу в маленькой гималайской деревушке на высоте около 2000 метров над уровнем моря, и главным образом с этого момента разработка пошла в гору. Это сократило темпы расхода нашей наличности, что очень важно для начала работы без посторонней помощи, и дало мне достаточно времени и спокойствия для работы над Neomailbox и Cryptonite.

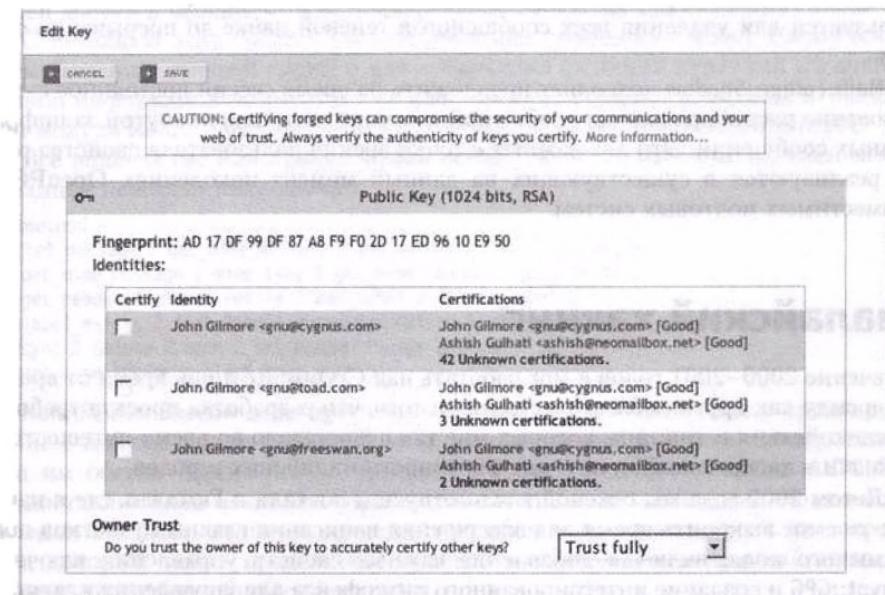


Рис. 11.5. Диалоговое окно Edit Key

Даже при том, что у нас были свои трудности в работе над высокотехнологичными системами для решения ответственных задач в отдаленной гималайской деревне, которая, по большому счету, еще жила в XIX столетии, слова Николая Переиха, русского художника, писателя и философа, который много лет

жил в той же части Гималаев, во многом и для нас сохранили свою актуальность: «По правде говоря, только здесь, только в Гималаях, есть уникальные, ни с чем не сравнимые по спокойствию условия для достижения результатов».

Организация защиты кода

Первоначально код был разработан в виде прототипа, и меня не слишком волновали вопросы его защиты. Но когда настало время сделать систему публичным достоянием и выпустить бета-версию, пришел черед и закрытию кода, для чего необходимо осуществить, как минимум, следующее:

- полное разделение привилегий;
- тщательную проверку входных данных;
- проверку защиты Crypt::GPG;
- документирование любых потенциальных проблем безопасности.

Разделение привилегий было встроено с самого начала, путем запуска `smaild` в качестве привилегированного пользователя и взаимодействия с ним через его API. Это дало возможность `smaild` выполнять такие привилегированные операции, как изменение конфигурационных файлов системы и выполнение криптографических операций методом контроллера, без предоставления процессу на веб-сервере доступа к жизненно важным ресурсам. Уточнение разделения полномочий между ядром и интерфейсом понадобилось лишь для нескольких областей.

Одной из таких областей была композиция MIME-сообщений с двоичными прикреплениями. Когда код создавался с использованием `Persistence::Object::Simple`, протокол `smaild` для двоичной композиции MIME-сообщения был обойден. Выкладываемые пользователем прикрепления сохранялись во временном каталоге, к которому имели доступ как `smaild`, так и процесс, происходящий на веб-сервере. Таким образом, `smaild` и веб-интерфейс `Cryptonite` нужно было размещать на одном и том же сервере.

С переходом на `Persistence::Object::Postgres` появилась возможность без труда передавать двоичные объекты между внешним интерфейсом и выходным буфером через базу данных, не полагаясь на прямые операции файловой системы. Это важно, потому что интерфейс, база данных и движок `Cryptonite` планировалось запускать на своих собственных, независимых серверах или на кластерах со сбалансированной загрузкой.

Добавка проверки входных данных (для проверки приемлемости предоставленных пользователем входных данных, таких как папка и идентификаторы сообщений) особого труда не представляла. Для добавления этой проверки к каждому методу `Cryptonite::Mail::Service` использовался слегка измененный модуль `Params::Validate`. К примеру, в методе `mvmgs` проверка его входных данных осуществлялась с помощью следующего кода:

```
sub mvmgs { # перемещение списка сообщений на какой-нибудь другой почтовый
    # ящик.
    my ($self, $username, $key, $dest, $copy, @msgnums) =
```

```

(shift, lc shift, shift);
my ($user, $session, $err) = $self->validateuser($username, $key);
return $err if $err;
return $self->cluebat(@{$@}) unless eval {
    ($dest, $copy, @msgnums) = validate_with ( params => \@_,
        extra => [$self], spec = [
            { type => SCALAR, callbacks =>
                { 'Legal Folder Name' => $self->legal_foldername } },
            { type => SCALAR, callbacks =>
                { 'Boolean Flag' => $self->opt_boolean }, optional => 1 },
            { type => SCALAR, callbacks =>
                { 'Legal Message Number' => $self->legal_msgnum } }
            x (@_ - 2)
        ]
    );
};

```

Приемлемость предоставленного пользователем ввода для каждого типа входного поля определена через ссылки на подпрограммы обратного вызова, сохраненные в хэше, в модуле Cryptonite::Mail::Config:

```

LGL_FOLDERNAME => sub { $_[0] =~ /$_[1]->{"VFOLDER"}/i
    or die ([{'EBADFOLDER'}, $_[0]]));
OPT_BOOLEAN => sub { $_[0] eq '' or $_[0] eq 0 or $_[0] eq 1
    or die ([{'EBADBOOL'}, $_[0]]));
LGL_MSGNUM => sub { $_[0] =~ /$_[1]->{"VMSGNUM"}/
    or die ([{'EBADMSGNUM'}, $_[0]]});

```

Подобные подпрограммы вызываются при каждой проверке входных параметров. Используемые при проверке регулярные выражения хранятся отдельно в Cryptonite::Mail::Config.

Хотя большинство проверочных подпрограмм, по существу, одинаковы, все они отделены друг от друга, чтобы по мере необходимости любой мог провести их тонкую настройку, не воздействуя на все остальные или не жертвуя ясностью в этой части кода. Используемые при проверке регулярные выражения и строки с описаниями ошибок также хранятся в таблице, чтобы в будущем можно было провести локализацию продукта.

Persistence::Object::Postgres также проводит свои собственные санитарные проверки, чтобы защититься от инъекционных SQL-атак.

Ревизия Crypt::GPG

Модуль Crypt::GPG был написан как рабочий прототип и нуждался в полной ревизии, чтобы исключить любую потенциальную проблему безопасности перед публичным тестированием системы.

Свободный доступ к Crypt::GPG был открыт в CPAN с 2001 года, и я получил множество ценных отзывов от его пользователей. Хотя многие пользователи сообщили о том, что они довольны простотой и ясностью имеющегося в модуле интерфейса, у некоторых из них возникли проблемы с его запуском на конкретных платформах, где модуль Expect, используемый им для взаимодействия с GnuPG, работал неправильно. (Expect использует в качестве своего IPC-механизма псевдотерминалы Unix [pty] и не работает, к примеру, под Windows.)

К тому же интерфейс и синтаксис модуля Expect были немного запутанными, что затрудняло чтение кода, о чем свидетельствует следующий фрагмент метода sign:

```
my $expect = Expect->spawn ($self->gpgbin, @opts, '-o-', '--sign',
                             @extras, @secretkey, $tmpnam);
$expect->log_stdout($self->debug);

$expect->expect (undef, '-re',
                  '----BEGIN', 'passphrase:', 'signing failed');

if ($expect->exp_match_number == 2) {
    $self->doze; print $expect ($self->passphrase . "\r");
    $expect->expect (undef, '-re', '----BEGIN', 'passphrase');

    if ($expect->exp_match_number == 2) { # Переданная фраза неверна
        $self->doze;
        print $expect ($self->passphrase . "\r");
        $expect->expect (undef, 'passphrase:'); $self->doze;
        print $expect ($self->passphrase . "\r");
        $expect->expect (undef);
        unlink $tmpnam;
        return;
    }
}

elsif ($expect->exp_match_number == 3) {
    unlink $tmpnam; $expect->close;
    return;
}

$expect->expect (undef);
my $info = $expect->exp_match . $expect->exp_before;
```

Использование модуля на основе Expect также приводило к так называемой гейзенберговской ошибке — к трудновоспроизводимому отказу, что, согласно моим исследованиям, являлось результатом слишком быстрой отправки входных данных в gpg. Вызовы doze в предыдущем коде обходили эту проблему: в них перед отправкой следующего бита входной информации в gpg вводятся несколько миллисекунд задержки. В большинстве случаев это срабатывало, но отказы на слишком загруженных системах все равно возникали.

Все эти проблемы указывали на необходимость отхода от Expect и использования другого механизма взаимодействия с двоичным кодом GnuPG. У меня был замысел целиком написать реализацию OpenPGP на Perl, но я принял другое решение, в первую очередь в основном по тем же причинам, по которым я решил использовать GnuPG: Cryptonite прежде всего является почтовым клиентом с интегрированной поддержкой OpenPGP. Полная реализация OpenPGP как минимум удвоила бы сложность кода, который мне нужно было бы обслуживать¹.

¹ Пополненная реализация OpenPGP на Perl, Crypt::OpenPGP, была написана Беном Троттом (Ben Trott) в 2001–2002 годах, и ее можно получить в CPAN. Я надеюсь использовать ее в будущих версиях Cryptonite, которые будут поддерживать несколько криптографических внутренних модулей.

После недолгого экспериментирования выяснилось, что для связи с GnuPG вполне может подойти IPC::Run, разработанный Барри Слэймейкером (Barrie Slaymaker). С IPC::Run предыдущий код приобрел следующий вид:

```
my ($in, $out, $err, $in_q, $out_q, $err_q);
my $h = start ([${self->gpgbin}, @opts, @secretkey, '-o-',
                '--sign', @extras, $tmpnam],
               \$in, \$out, \$err, timeout( 30 ));

my $i = 0;

while (1) {
    pump $h until ( $out =~ /NEED_PASSPHRASE (.{16}) (.{16}).*\n/g or
                    $out =~ /GOOD_PASSPHRASE/g);
    if ($2) {
        $in .= ${self->passphrase} . "\n";
        pump $h until $out =~ /(GOOD|BAD)_PASSPHRASE/g;
        last if $1 eq 'GOOD' or $i++ == 2;
    }
}
finish $h;
my $d = $detach ? 'SIGNATURE' : 'MESSAGE';
$out =~ /(^-----BEGIN PGP $d-----.*-----END PGP $d-----)/s;
my $info = $1;
```

IPC::Run надежно работает и без минизадержек, необходимых для Expect, намного проще читается и отлично работает на большинстве платформ.

Некоторые операции с gpg не требовали никакого взаимодействия, и в ранней версии модуля для таких случаев использовались операторы Perl, взятые в обратные апострофы (backtick operator). Поскольку такие операторы вызывают оболочку, возникает риск безопасности. При использовании IPC::Run можно было легко заменить backtick-операторы более безопасной функцией backtick, обходя таким образом обращение к оболочке. Это дает возможность исключить в Crypt::GPG вызовы оболочки.

```
sub backtick {
    my ($in, $out, $err, $in_q, $out_q, $err_q);
    my $h = start (@_, \$in, \$out, \$err, timeout( 10 )):
    local $SIG{CHLD} = 'IGNORE';
    local $SIG{PIPE} = 'IGNORE';
    finish $h;
    return ($out, $err);
}
```

Некоторые пользователи также указали на то, что использование временных файлов для хранения открытого текста может тоже представлять угрозу безопасности. Эта проблема могла быть легко преодолена без вмешательства в код, просто за счет использования виртуального диска с шифрованием путем перестановки (например предоставляемого OpenBSD) или зашифрованного виртуального диска, при этом открытый текст никогда бы не записывался на диск в незашифрованном виде.

Разумеется, было бы неплохо модифицировать код, чтобы вообще избежать записи открытого текста во временные файлы, но поскольку уже было практическое средство обхода, устранение временных файлов было отложено на будущее, а не исполнено немедленно.

Новый Crypt::GPG, основанный на IPC::Run, был выложен в CPAN в конце 2005 года. Теперь он работал в более широком диапазоне операционных систем, был более надежен и безопасен, чем его основанный на Expect предшественник.

Скрытые манипуляции

К середине 2004 года Neomailbox исполнился год, и он привлек довольно много рассчитывающих за него клиентов. Разработка Cryptonite была на некоторое время отложена, поскольку мне нужно было поработать над различными аспектами сервиса Neomailbox, кроме того, нужно было приступать к другим, не терпящим отлагательства проектам.

Но то, что продукт был выпущен на рынок, сыграло весьма позитивную роль, поскольку в дело включились рыночные силы, от борьбы за пользовательские отзывы до влияния на процесс разработки, и помогло выявить и уточнить приоритеты. Требования и запросы клиентов позволяли мне глубоко вникать в суть пользовательских и рыночных потребностей.

Удовлетворение рыночных потребностей делает код приложения привлекательным в коммерческом смысле. В конечном счете, подобное взаимодействие с рынком становится неотъемлемым и решающим компонентом процесса разработки.

Cryptonite был сконструирован в расчете на легкость поддержки и модификации, именно потому, что я знал, что в какой-то момент он должен будет начать развитие в новых направлениях, отвечая как на требования, так и на ожидания клиентов. Присутствие на рынке позволило мне отслеживать возникающие запросы: стало ясно, что будущее в удаленном доступе к почтовым ящикам будет за IMAP.

IMAP обладает массой привлекательных свойств, которые делают его очень мощным и действенным протоколом доступа к электронной почте. Одним из важнейших свойств является возможность доступа к одному и тому же почтовому ящику с использованием нескольких клиентов, значение которого с быстрым ростом числа компьютерных устройств неизменно возрастает. Сейчас у обычного пользователя есть настольный компьютер, ноутбук, наладонник и сотовый телефон, и все они могут получить доступ к его почтовому ящику.

В связи с этим возникла небольшая проблема, поскольку я уже завершил создание хранилища электронной почты для Cryptonite, основой которого IMAP не был. Из этой ситуации было два выхода: либо реализовать на основе хранилища электронной почты Cryptonite полноценный IMAP-сервер (что было довольно объемной работой), либо модифицировать Cryptonite, дав ему возможность использовать в качестве внутренней структуры хранилище электронной почты на основе IMAP. На самом деле все равно пришлось бы остановиться на втором варианте.

И опять, стремясь к уменьшению сложности системы и концентрируясь на ее первоначальном предназначении, я решил не разрабатывать почтовое хранилище Cryptonite внутри полномасштабного IMAP-сервера. Вместо этого я модифицировал его, превратив в механизм кэширования, который кэшировал

MIME-структуры (только структурную информацию без контекста) составных MIME-сообщений, перечисленных пользователем, а также целиком все читаемые пользователем сообщения, поэтому когда пользователь еще раз открывал ранее прочитанное сообщение, Cryptonite не нуждался в возврате к IMAP-серверу для его повторного извлечения.

Это позволило мне взять все самое лучшее от обоих вариантов.

В программе Cryptonite могло отражаться содержимое почтового ящика IMAP, и одновременно с этим она располагала полной и точной информацией о МЕМО-структуре каждого сообщения, наряду с возможностью сохранять доступность расшифрованных сообщений в теневых папках, поддерживаемых почтовым хранилищем этой программы.

Модификация кода не вызывала затруднений. Как только пользователь щелкал мышью для чтения сообщения, которого не было в кэше, Cryptonite кэшировал его в соответствующей папке Mail::Folder::Shadow:

```
my $folder = $session->folder; # Имя папки
my $mbox = _opencache($username, $folder); # Кэш Mail::Folder::Shadow

unless ($msgnum and grep { $_ == $msgnum } $mbox->message_list) {

    # Сообщение не кэшировано. Извлечь его из IMAP-сервера и кэшировать.

    my $imap = $self->_open_imapconn($username, $user->password)
        or sleep(int(rand(3))+2), return $self->cluebat (EBADLOGIN);

    $imap->select($folder) or return $self->cluebat (ENO_FOLDER, $folder);

    $imap->Uid(1);

    my ($tmpfh, $tmpnam) =
        tempfile( $self->tmpfiles, DIR => "$HOME/tmp",
                  SUFFIX => $self->tmpsuffix, UNLINK => 1);
    $imap->message_to_file($tmpfh, $msgnum);

    $imap->logout;

    my $parser = new MIME::Parser; $parser->output_dir("$HOME/tmp/");

    $parser->filer->ignore_filename(1); # Не использовать предлагаемое имя
                                         # файла

    seek ($tmpfh, 0, 0);
    my $mail = $parser->parse($tmpfh);

    return $self->cluebat (ENOSUCHMSG, 0 + $msgnum) unless $mail;
    $mbox->update_message($msgnum, $mail);
}
```

Примерно также для каждого сообщения, отмеченного пользователем в перечне сообщений, кэшируется и MIME-структура. Весь остальной код работает как и прежде, задействуя кэш при всех операциях чтения. Теперь мы получили

IMAP-совместимость, не поступившись функциональностью моего хранилища электронной почты и не проводя глубокой модификации основного кода.

Дублирование почтового хранилища нужно было разрабатывать заново, поскольку переход для этого хранилища с `Mail::Folder::SQL` на IMAP-сервер означал, что `Replication::Recall` не может быть использован для дублирования. Но в любом случае, `Replication::Recall` не был самой элегантной или легкой в осуществлении системой дублирования, и библиотека `Recall` была переписана на языке Python, и мой, написанный на Perl интерфейс к ранней реализации на C++, все равно стал не нужен.

Получилось, что я непредусмотрительно затратил слишком много времени на разработку функции дублирования, которую пришлось выкинуть, и наверное мне лучше было бы на этой стадии вообще не заниматься дублированием. С другой стороны, я научился многому тому, что мне пригодилось бы при переходе к осуществлению нового дублирования.

Рыночные силы и изменяющиеся стандарты приводят к тому, что прикладное программное обеспечение все время совершенствуется, и с точки зрения программиста красота такого программного кода, безусловно, в том, насколько просто этот код можно приспособить к постоянно изменяющимся требованиям. Объектно-ориентированная архитектура Cryptonite дает возможность легко реализовывать весьма существенные переработки.

Скорость тоже имеет значение

С использовавшимся в Cryptonite хранилищем электронной почты производительность была на весьма высоком уровне, и большинство операций хранения электронной почты не зависели от размера почтового ящика. Но переключившись на IMAP, я заметил значительное замедление работы с большими почтовыми ящиками. Небольшой анализ показал, что узким местом в производительности был полностью написанный на Perl модуль `Mail::IMAPClient`, которым я пользовался для осуществления IMAP-совместимости.

Наскоро написанный сценарий сравнения производительности (использовавший модуль `Benchmark`) помог мне проверить, нет ли другого подходящего CPAN-модуля, и `Mail::Cclient`, обеспечивающий интерфейс с библиотекой UW C-Client, был более эффективным, чем `Mail::IMAPClient`. Результаты однозначно показывали, что мне нужно переделать IMAP-код, используя `Mail::Cclient`:

	Rate	IMAPClientSearch	CclientSearch
IMAPClientSearch	39.8/s	--	-73%
CclientSearch	145/s	264%	--
	Rate	IMAPClientSort	CclientSort
IMAPClientSort	21.3/s	--	-99%
CclientSort	2000/s	9280%	--

Наверное, я должен был подумать о тестировании производительности нескольких разных модулей, прежде чем написать код с использованием `Mail::IMAPClient`. Сначала я избегал использования библиотеки C-Client, поскольку

совсем не хотел усложнять процесс разработки, а он был намного проще с Mail::IMAPClient, чем с Mail::Cclient.

К счастью, переход от старого к новому прошел без осложнений. Я заметил, что IMAPClient не мог выполнять некоторые операции лучше, чем C-Client, без существенной потери производительности, поэтому теперь в Cryptonite::Mail::Service используются оба модуля, каждый из которых делает то, что у него лучше получается.

Программы, подобные Cryptonite, конечно, никогда не будут в полной мере «готовы», но теперь код стал более сформировавшимся, надежным, полноценным и достаточно эффективным для того, чтобы отвечать своему предназначению: предоставлять тысячам одновременно работающим пользователям безопасную, интуитивно понятную, легкую в управлении электронную почту, которая помогает им эффективно защитить конфиденциальность связи.

Конфиденциальность связи для обеспечения прав человека

В начале этой главы я отметил, что распространение повсеместно доступной технологии обеспечения конфиденциальности связи является очень эффективным средством для защиты прав человека. Поскольку осознание этого обстоятельства стало основной мотивацией для разработки проекта Cryptonite, я хотел бы закончить главу некоторыми наблюдениями на этот счет.

- Кроме всего прочего технология Cryptographic способна на следующее¹:
- предоставляет мощную, спасающую жизнь защиту для активистов, членов неправительственных организаций и репортеров, работающих в странах с репрессивными режимами²;
 - сохраняет возможность передачи подвергаемых цензуре новостей и спорных мыслей;
 - защищает анонимность осведомителей, свидетелей и жертв домашнего насилия;
 - и на десерт, является катализатором создания общества, не разделенного расстояниями, предоставляя возможность свободного и беспрепятственного обмена идеями, ценностями и услугами по всему земному шару.

Разношерстная команда хакеров, известная как киберпанки, разрабатывала программное обеспечение, способствующее повышению уровня конфиденциальности в течение многих лет, стремясь к усилению личной свободы и независимости в цифровую эпоху. Часть криптографического программного обеспечения уже стала краеугольным камнем всей сегодняшней работы в мире. К нему относится программа удаленного терминала Secure SHell (SSH), являющаяся основным средством обеспечения безопасности инфраструктуры Интернета,

¹ См. <http://www.idiom.com/~arkuat/consent/Anarchy.html> и <http://www.chaum.com/articles/>.

² См. <http://philzimmermann.com/EN/letters/index.html>.

и шифровальный набор Secure Sockets Layer (SSL), обеспечивающий безопасность интернет-торговли.

Но эти системы предназначены для удовлетворения весьма специфических запросов: соответственно, безопасности доступа к серверу и защите проводящихся через Интернет транзакций с использованием кредитных карт. Обе эти системы связаны с безопасностью взаимодействия на уровне человек-машина. Чтобы противостоять возрастающей угрозе тотального наблюдения (что «ведет к быстрому закату цивилизации»¹), в ближайшие годы должно быть развернуто значительное количество средств применения криптографических технологий в сфере чисто человеческого общения.

Простая в использовании, безопасная система электронной почты является многообещающей технологией — она позволяет впервые в истории иметь безопасную и конфиденциальную удаленную связь между людьми по всему земному шару, которым никогда не понадобится для этого личная встреча.

Хакинг цивилизации

Столь бесконечно сложный и тонкий компьютер, как Земля, которая включает в свою функциональную матрицу органическую жизнь, и цивилизация, представляющая собой ее узлы, — могут быть весьма действительно перепрограммированы простым фрагментом кода, который опять приведет к тонким изменениям человеческой культуры, переделывая операционную систему самого общества.

Код приводил к изменениям мира множество раз. Стоит только взглянуть на успехи медицины, ставшие возможными за счет программного обеспечения для расшифровки генетической последовательности, на влияние программ для ведения бизнеса крупных компаний, а также мелких предприятий, на революцию, вызванную программным обеспечением, относящимся к автоматизации производства и компьютерному моделированию, или на множество революций, связанных с Интернетом: электронная почта, как таковая Всемирная сеть, блоги, службы социальных сетей, VoIP... Становится понятным, что современная история во многом складывается из истории программных инноваций.

Конечно, код, как и любая технология, может быть обоюдоострым, либо ускоряя «возврат к насилию»² в обществе, либо его замедляя, повышая эффективность технологий вмешательства в частную жизнь и предоставляя тиранам более действенные инструменты цензуры с одной стороны или укрепляя права человека и содействуя их соблюдению с другой. Любая разновидность кода приводит к непосредственным преобразованиям самих основ человеческого общества, изменяя основополагающую социальную действительность, такую как способность защитить свободу слова.

¹ Vernor Vinge, «A Deepness in the Sky». Tor Books, 1999.

² Под «возвращением к насилию» я понимаю социальные и экономические стимулы нарушения прав человека. Авторы книги о полноправии человека — «The Sovereign Individual» — отмечали: «Ключ к пониманию развития общества лежит в понимании факторов, определяющих издергки и вознаграждения применяющего насилие».

Интересно, что даже такая отдельно взятая технология, как криптография, для реализации которой выбран открытый ключ, может существенно изменить культурную действительность. К примеру, реализация на основе инфраструктуры открытого ключа (PKI) повторно налагает разрешительные свойства, такие как централизованная иерархия и идентификационные требования, на технологию, вся ценность которой, возможно, заключается в отсутствии в ней этих свойств. Несмотря на, это подходы, использующие PKI, предоставляют более слабую ключевую аутентификацию, чем реализация, использующая сети доверия (которая к тому же не ослабляет другие важные свойства криптографии с открытым ключом, такие как рассредоточенное развертывание).

Как создатель кода, я считаю, что по большому счету этическая обязанность программистов состоит в том, чтобы стремиться к красоте кода не только в его конструкции и реализации, но и в тех результатах, которые он принесет в широком социальном контексте. Вот почему я считаю освобождающий код таким красивым. Он возводит вычислительную технологию на высоты самого благородного использования: для защиты человеческих прав и человеческой жизни.

Законы и международные соглашения по правам человека распространяются только на защиту индивидуальных прав. История показывает, что их очень легко обойти или проигнорировать. С другой стороны, математика криптографических систем при ее качественной реализации может обеспечить фактически непробиваемую защиту индивидуальных прав и открытых выражений мысли и может, наконец, позволить людям во всем мире свободно связываться и конфиденциально и свободно общаться.

Реализация глобального, математически защищенного открытого общества во многом зависит от нас, повелителей машин.

12 Становление красивого кода в BioPerl

Линкольн Стейн (*Lincoln Stein*)

За прошедшее десятилетие биология превратилась в информационную науку. Новые технологии дают биологам возможность беспрецедентных проникновений в мир загадочных процессов, протекающих внутри клеток животных и растений. Машины, устанавливающие последовательность ДНК, дают возможность быстрого считываивания детализированных последовательностей генома; микроматричные технологии предоставляют отображения мгновенного состояния сложных структур генной формулы в развивающихся организмах; конфокальные микроскопы создают трехмерные фильмы для отслеживания изменения в клеточной структуре при превращении предраковых тканей в злокачественные образования.

Эти новые технологии обычно приводят к генерации терабайтов данных, и все эти данные должны быть отфильтрованы, сохранены, соответствующим образом обработаны и отобраны. Информатика и разработка программного обеспечения применительно к проблемам обработки биологических данных называется *биоинформатикой*.

Во многих отношениях биоинформатика похожа на разработку программного обеспечения для Уолл-стрит. Разработчики в сфере биоинформатики, как и программисты, работающие на финансовый сектор, должны быстро реагировать на изменения: им приходится создавать приложения в высоком темпе, затрачивая минимальное время на анализ требований и проектирование. Наборы данных, с которыми они работают, весьма объемны и изменчивы, их срок годности измеряется не годами, а месяцами. Поэтому многие разработчики программ биоинформатики предпочитают гибкие технологии, вроде экстремально-го программирования (eXtreme Programming), и инструментарий для быстрой разработки прототипов и ввода программ в действие. Здесь, как и в финансово-м секторе, уделяется большое внимание визуализации данных и распознаванию образов.

BioPerl и модуль Bio::Graphics

Одним из средств быстрой разработки, созданных специалистами биоинформатики для своих же нужд, является BioPerl – обширная, общедоступная библиотека многократно применяемого кода для языка программирования Perl. BioPerl предоставляет модули для решения наиболее распространенных биоинформационных задач, в которых используется анализ ДНК и белковых структур, построение и анализ эволюционных древовидных структур, расшифровка генетических данных и, конечно же, анализ генетических последовательностей.

BioPerl позволяет программисту быстро создавать сложные конвейеры по обработке, фильтрации, анализу, объединению и визуализации больших наборов биологических данных. Благодаря интенсивному тестированию библиотеки сообществом пользователей открытого кода созданные на основе BioPerl приложения имеют высокую вероятность заработать с первого раза, а поскольку интерпретаторы Perl доступны на всех основных платформах, приложения, включающие модули BioPerl, будут работать на машинах под управлением Microsoft Windows, Mac OS X, Linux и Unix.

В этой главе рассматривается Bio::Graphics, модуль BioPerl, предназначенный для визуализации карты генома. Он решает задачу визуализации и комментирования генома, который состоит из набора ДНК-последовательностей, каждая из которых представляет собой буквенную строку [A,G,C,T] нуклеотидов, известных также как *пары нуклеотидов* (base pairs – *bp*). Некоторые строки ДНК-последовательностей могут иметь весьма значительную протяженность: например, геном человека состоит из 24 строк ДНК-последовательностей, по одной для хромосом с 1 по 22, плюс по одной для Х и Y хромосом. Длина самой длинной из них, хромосомы № 1, приблизительно равна 150 000 000 bp (150 миллион пар нуклеотидов).

Спрятанные внутри этих ДНК-последовательностей многочисленные участки играют свою роль в клеточном метаболизме, репродукции, защитных реакциях и передаче сигналов. Например, некоторые участки хромосомы № 1 ДНК-последовательности являются генами, кодирующими белок. Эти гены «транскрибируются» клеткой в короткие РНК-последовательности, которые перемещаются из клеточных ядер в цитоплазму; затем эти РНК-последовательности транслируются в белок, ответственный за генерацию энергии, за перемещение питательных веществ в клетку и из нее, за создание клеточной мембранны и т. д. Другие участки ДНК-последовательности служат естественными регуляторами: когда белок-регулятор привязывается к определенному месту регулирования, близлежащий ген, кодирующий белок, «включается», и начинается его транскрибирование.

Некоторые участки представляют собой паразитическую часть ДНК: короткие участки последовательности, которые могут полуавтономно копировать сами себя и перебираться по геному с места на место. Назначение некоторых других участков неизвестно; можно сказать, что они тоже имеют важное значение, потому что были сохранены в человеческих и других организмах на протяжении продолжительных эволюционных интервалов, но их предназначение еще не выяснено.

Поиск и интерпретация функционально значимых участков генома называется *аннотацией* и является главной задачей, на которой сосредоточивается проект его исследования. При аннотации генома обычно генерируется гораздо больше данных, чем представлено в самой ДНК-последовательности. Полная генетическая последовательность человека занимает в незархивированном виде три гигабайта памяти, но его текущая аннотация использует многие терабайты (см. также главу 13).

Пример выходной информации Bio::Graphics

Чтобы сосредоточиться на «интересном» участке генома, биологам нужно точно представить, как множество аннотаций соотносятся друг с другом. Например, предполагаемый участок регулирования, вероятнее всего, будет функционально значимым, если будет располагаться поблизости к кодирующему белок гену и перекрываться участком, сохраненным между эволюционно отдаленными разновидностями.

Bio::Graphics дает возможность программам биоинформатики осуществлять быстрое отображение генома и его аннотаций. Он может быть использован автономно для генерации статического изображения участка в различных графических форматах (включая PNG, JPEG и SVG) или встраиваться в обычное или веб-приложение, обеспечивая интерактивную прокрутку, изменение масштаба и изучение данных.

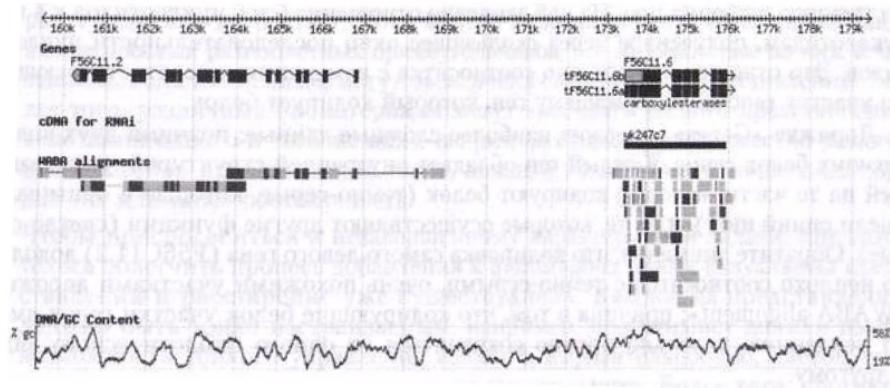


Рис. 12.1. Пример изображения, сгенерированного модулем Bio::Graphics

На рис. 12.1 показан пример изображения, сгенерированного модулем Bio::Graphics. На изображении показан участок генома свободноживущей нематоды (небольшого земляного черва), который иллюстрирует некоторые аспекты обычной картинки, генерируемой Bio::Graphics. По вертикали изображение разделено на ряд горизонтальных дорожек. Верхняя дорожка состоит из горизонтальной

шкалы, идущей слева направо. Она проградуирована в килобазах («k»), обозначающих тысячи нуклеотидных пар ДНК. Показанный участок начинается непосредственно перед позицией 160 000 хромосомы I свободноживущей нематоды, и простирается за позицию 179 000, охватывая в общей сложности 20 000 нуклеотидных пар. На рисунке представлены четыре аннотационные дорожки, каждая из которых иллюстрирует возрастающее по сложности зрительное отображение.

Оригинальное изображение имеет яркую раскраску, но напечатано в серых тонах. Самая простая дорожка «cDNA for RNAi» показывает позиции образца экспериментального реагента, созданного исследовательским сообществом для изучения организации генов свободноживущей нематоды. В правой части изображения находится отдельная аннотация, названная $\text{yk}247\text{c}7$. Она состоит из черного прямоугольника, который начинается приблизительно с позиции 173 500 и простирается до позиции 176 000, и соотносится с физической частью ДНК, охватывающей этот участок, который исследователь может заказать у компании, поставляющей биоматериалы, и использовать для экспериментов для изменения активности перекрываемого им гена, в данном случае – F56C11.6.

Дорожка «WABA alignments» отображает немного более сложную информацию. Она дает наглядное представление о количественных данных, полученных в результате сравнения этой части генома свободноживущей нематоды с такими же участками другого червя. Очень похожие участки окрашены в темно-серые тона. Менее похожие участки окрашены в светло-серые тона. Участки со средней степенью схожести окрашены в умеренно-серые тона.

Дорожка «DNA/GC Content» показывает постоянно изменяющуюся количественную информацию. На ней записано отношение G и C нуклеотидов к A и T нуклеотидам, полученное через скользящее окно последовательности нуклеотидов. Это отношение примерно соотносится с изменениями, соответствующими участку генома, содержащему ген, который кодирует белок.

Дорожка «Genes» содержит наиболее сложные данные: позиции двух кодирующих белок генов. Каждый ген обладает внутренней структурой, указывающей на те части, которые кодируют белок (темно-серые, которые в оригинале имели синий цвет), и на те, которые осуществляют другие функции (светло-серые). Обратите внимание, что кодировка самого левого гена (F56C11.2) довольно неплохо соотносятся с темно-серыми, очень похожими участками дорожки «WABA alignment»; причина в том, что кодирующие белок участки генов имеют тенденцию довольно прочно сохраняться от одного биологического вида к другому.

Ген по имени F56C11.6 аннотирован в соответствии с функциональным предназначением («carboxylesterases» – карбоксилистеразы), показывающим его отношение к семейству белков, ответственных за основную часть углеродного метаболизма. Кроме того, онображен в двух альтернативных формах, показывающих, что он может кодировать более одного отдельного белка. Эти две альтернативные формы сгруппированы вместе и снабжены отдельными надписями. Обратите внимание на многочисленные раскладки, расположенные ниже этого гена; они представляют собой отражение того факта, что ген

принадлежит большому семейству родственных генов, и каждый родственный ген представлен отдельной раскладкой.

Реально существующая ДНК-последовательность нуклеотидов в этом представлении не показана, поскольку уместить в 800 пикселях 20 000 нуклеотидных пар физически невозможно. Тем не менее при отображении с помощью Bio::Graphics более короткого участка генома могут быть нарисованы фактические буквенные обозначения с соответствующим оформлением (например с изменением их цвета или подсветки), чтобы показать начальные и конечные позиции интересующих деталей.

Требования, предъявляемые к Bio::Graphics

Работа Bio::Graphics заключается в получении серий аннотаций генома (которые терминах BioPerl называются свойствами – *features*) и выдаче графического файла, отформатированного в соответствии с заданными программисту техническими требованиями. У каждого свойства есть начальная и конечная позиция и направление (указывающее налево, направо или не указывающее ни в одном из направлений). Свойства могут быть связаны с другими свойствами, такими как имя, описание и числовое значение. Свойства также могут иметь внутреннюю структуру и содержать подсвойства и подподсвойства.

Проектируя эту библиотеку, я должен был обратиться к решению следующих вопросов.

Незавершенный характер проблемы

Существует огромное количество видов геномных аннотаций, число которых ежедневно возрастает. Хотя многие аннотации могут быть нарисованы в виде простых разноцветных прямоугольников, все же некоторые из них, в частности, количественные, могут быть очень сложными в представлении. Более того, различные биоматериалы могут требовать разного представления аннотаций одного и того же вида; например, существует множество различных способов представления генов, каждый из которых больше подходит для тех или иных обстоятельств.

Чтобы приспособиться к незавершенному характеру этой задачи, мне захотелось облегчить процесс добавления к Bio::Graphics новых визуальных представлений и расширение уже существующих. Каждое из представлений должно быть тонко настраиваемым; например, программист должен иметь возможность тонкого управления высотой, выразительностью, цветом границ и заливки каждого отдельного прямоугольника. Более того, программист должен иметь возможность изменять вариант каждого свойства в зависимости от конкретного случая.

Плотность свойств

Некоторые геномные свойства обладают большой плотностью, тогда как другие имеют более разреженные свойства (сравните дорожки «Genes» и «WABA alignments» на рис. 12.1). Свойства могут также пространственно накладываться друг на друга. Иногда наложенные друг на друга свойства

вам нужно рассмотреть по отдельности, а иногда нужно посмотреть на наложения, смешая наложенные свойства вверх-вниз по вертикали, отделяя их друг от друга. Чтобы быть добавленными к задаче, свойства должны содержать подсвойства, которые непосредственно накладываются друг на друга, поскольку просмотр наложений и пространственная схема размещения должны работать рекурсивно.

Я считаю, что было бы неплохо, чтобы просмотр наложений мог быть активирован и dezактивирован контекстно-зависимым способом. Например, если в отображаемом участке имеются тысячи перекрывающихся свойств, просмотр наложений может привести к тому, что дорожка станет неуправляемой из-за перегруженности, и потребовать отключения. Программист в таком случае должен иметь возможность в зависимости от контекста управлять просмотрами наложений, приводящими к краху, или полностью их отменять.

Масштабирование

Мне хотелось, чтобы Bio::Graphics был в состоянии нарисовать очень подробную картинку участка генома в 500 bp, а также отобразить весь геном, состоящий из 150 миллионов нуклеотидов. Чтобы решить эту задачу, визуальное отображение свойств должно разумно изменяться в соответствии с текущим масштабом. При масштабе в 500 bp можно отобразить внутреннюю структуру гена. При масштабе в 1 000 000 bp можно отобразить только начальные и конечные позиции гена. При масштабе в 100 000 000 bp гены сливаются в единую черную полосу, и нужно перейти на представление, которое показывает плотность гена в полутонах.

Возможность использования в интерактивных веб-приложениях

Я хотел, чтобы Bio::Graphics подходил к выходному буферу интерактивного веб-браузера для просмотра генома. В частности, я хотел, чтобы конечные пользователи имели возможность щелкать на графических исполнениях свойств, чтобы вызывать ниспадающие меню, ссылки на другие веб-страницы, просматривать всплывающие подсказки и т. д. Для воплощения всего этого в жизнь Bio::Graphics должен работать относительно быстро, чтобы генерировать изображения в реальном масштабе времени. Он также должен отслеживать позицию каждого из представленным им свойств (и по необходимости их подсвойств) в целях создания карты изображения, которую можно вернуть программисту.

Независимость от графических форматов

Я хотел, чтобы Bio::Graphics был способен генерировать соответствующие качеству экрана изображения низкого разрешения, которые можно было бы встраивать в веб-страницы, а также генерировать изображения высокого разрешения, обладающие типографским качеством. Для этого та часть Bio::Graphics, которая обрабатывает логику схемы размещения, должна быть отделена от той части, которая генерирует графику. В идеале должна быть реализована возможность выводить как растровые, так и векторные графические форматы.

Независимость от схем баз данных

Сообщество разработчиков программ биоинформатики сконструировало десятки форматов баз данных, предназначенных для управления данными аннотаций генома, от простых однородных файлов до замысловатых реляционных баз данных. Чтобы обеспечить максимальную универсальность, я хотел избежать связи Bio::Graphics с какой-нибудь определенной схемой базы данных. Должно быть почти одинаково просто вызвать Bio::Graphics для представления участка генома, описание которого находится в однородном файле, и заставить его представить участок генома, который описан в базе данных Oracle.

Процесс проектирования Bio::Graphics

Я не приверженец формального проектирования; вместо этого я обычно выписываю небольшие отрывки псевдокода, в котором содержится описание желаемого хода какой-нибудь работы («история кода»), немного манипулирую входными и выходными форматами, немного программирую и, если я не удовлетворен взаимодействием элементов системы, возвращаюсь назад и переделываю историю кода. Для чего-нибудь более серьезного, чем игровые приложения, я разрабатываю небольшие фрагменты системы и тестирую их в автономных программах, перед тем как решить, продвигаться ли дальше в этой части конструкции. Я храню свои заметки в виде «внутреннего монолога» в текстовом файле и часто сохраняю код в своем CVS-хранилище. Я пытаюсь сделать весь код визуально-привлекательным и простым. Если он не отличается простотой, значит, что-то в конструкции не так, и я возвращаюсь к «чертежной доске».

Проектирование порядка взаимодействия разработчика с модулем

Моей первой конструкторской задачей было определение порядка работы типичного приложения, использующего модуль Bio::Graphics. Сначала я составил историю кода, показанную в примере 12.1.

Пример 12.1. Основная история использования BioPerl::Graphics (псевдокод)

```
1 use Bio::Graphics::Panel;
2 @first_set_of_features = get_features_somewhow();
3 @second_set_of_features = get_more_features_somewhow();
4 $panel_object = Bio::Graphics::Panel->new(panel_options...);
5 $panel_object->add_track(@first_set_of_features,
                           track_options...);
6 $panel_object->add_track(@second_set_of_features,
                           track_options...);
7 print $panel_object->png;
```

История кода начинается с ввода основного класса объектов Bio::Graphics — Bio::Graphics::Panel (строка 1). Я полагал, что в этом объекте будут храниться выбранные параметры конфигурации для изображения в целом, такие как размерности выводимой диаграммы и ее масштаб (в пиках на пиксел), и при взаимодействии с пользователем этот объект будет основным.

История кода продолжается двумя вызовами на выборку массивов, содержащих свойства последовательностей (строки 2–3). Чтобы обеспечить независимость от базы данных, в которой хранятся свойства, я решил, что Bio::Graphics будет работать с перечнем свойств, которые уже были извлечены из базы данных. С моему удовлетворению, BioPerl уже поддерживал множество аннотационных баз данных через превосходный универсальный интерфейс. Свойство последовательности представляется интерфейсом под названием Bio::SeqFeatureI, который определяет набор методов, поддерживающих все свойства последовательностей. Большинство этих методов просты и понятны; например, \$feature->start() получает начальную позицию свойства, \$feature->end() получает конечную позицию свойства, а \$feature->get_SeqFeatures() получает все подсвойства. Для извлечения свойства из базы данных в BioPerl есть интерфейс, названный Bio::Seq-Feature::CollectionI, который предоставляет стандартный API для последовательного или произвольного извлечения свойств посредством запросов.

Затем в истории кода вызывается метод Bio::Graphics::Panel->new() (строка 4), создающий новый объект панели, и дважды вызывается метод add_track() (строки 5–6). С их помощью добавляются две дорожки, по одной для первого и второго набора свойств. При каждом вызове add_track() берется начальный аргумент, состоящий из ссылки на добавляемый массив свойств, за которым следуют дополнительные аргументы, управляющие появлением дорожек.

Последним шагом в истории кода стоит преобразование панели в PNG-файл и немедленный вывод результата на стандартное устройство.

Сначала все было похоже на вполне разумную историю, но дальнейший анализ показал, что в ней есть несколько существенных недостатков в использовании API. Самая большая проблема состояла в том, что история заставляла программистов загружать все свойства в память перед началом формирования изображения. Тем не менее зачастую было бы удобнее считывать свойства последовательностей из файла или базы данных по одной. Некоторые особенности реализации Bio::SeqFeature::CollectionI позволяют вам это сделать:

```
$iterator = Bio::SeqFeature::CollectionI->get_seq_stream(@query_parameters);
while ($feature = $iterator->next_seq) {
    # какие-то действия со свойствами
}
```

Другая проблема состояла в том, что как только происходил вызов \$panel->add_track(), вы уже не могли изменить установки конфигурации дорожек. А я мог предположить ситуацию, при которой разработчику захотелось бы заранее добавить группу дорожек, а затем вернуться и изменить ранее установленную для них конфигурацию.

В конечном счете, мне показалось, что метод add_track() не отличался особой гибкостью, поскольку он вынуждал пользователей добавлять дорожки

в строгом порядке (сверху вниз). А должен был существовать еще и способ вставки дорожек в произвольном порядке.

Эти размышления навели на создание объекта Track:

```
$track1 = $panel_object->add_track(\@set_of_gene_features,
                                    track_options...);
$track2 = $panel_object->add_track(\@set_of_variation_features,
                                    track_options...);
```

После этого каждый мог бы добавить свойства к уже существующей дорожке:

```
$track1->add_features(feature1, feature2, feature3,...)
```

или вносить изменения в ее конфигурацию:

```
$track1->configure(new_options)
```

Это привело к другому варианту истории кода, показанному в примере 12.2.

Пример 12.2. История использования BioPerl::Graphics с учетом изменения дорожек (псевдокод)

```
1 use Bio::Graphics::Panel;
2 $panel_object = Bio::Graphics::Panel->new(panel_options...);
3 $track1 = $panel_object->add_track(bump_true, other_track_options...);
4 $track2 = $panel_object->add_track(bump_true, other_track_options...);

5 $collection = Bio::SeqFeature::CollectionI->new(@args);
6 $iterator = $collection->get_seq_stream(@query_parameters);

7 $genes=0; $variations=0;
8 while ($feature = $iterator->next_seq) {
9     if ($feature->method eq 'gene') {
10         $track1->add_feature($feature);
11         $genes++;
12     } elsif ($feature->method eq 'variation') {
13         $track2->add_feature($feature);
14         $variations++;
15     }
16 }
17 $track1->configure(bump_false) if $genes > 100;
18 $track2->configure(bump_false) if $variations > 100;

19 print $panel_object->png;
```

В этой версии истории мы сначала создаем две дорожки, не вкладывая в них заранее никаких свойств (строки 3–4). Тем не менее мы предоставляем методу add_track() ряд параметров, включая параметр «столкновений» – bump, который изначально установлен в true. При составлении этой истории я полагал, что параметр bump при установке в true будет активизировать контроль конфликтов, а при установке в false будет его отключать.

Потом с помощью BioPerl мы создаем объект семейства свойств; обычно это делается за счет подключения к какой-нибудь базе данных и запуска запроса

(строки 5–6) с возвращением итератора по результатам его выполнения. Затем мы многократно вызываем `$iterator->next_seq` для возвращения всех свойств, соответствующих условиям запроса (строки 8–16). Для каждого возвращенного свойства мы запрашиваем его тип, используя метод из `Bio::SeqFeatureI` под названием `method`. Если свойство относится к типу генов — `gene`, мы добавляем его к первой дорожке — `track1` и увеличиваем показание счетчика. Если свойство относится к типу разновидностей — `variation`, мы добавляем его ко второй дорожке — `track2` и увеличиваем показание другого счетчика.

После добавления всех свойств мы запрашиваем количество добавленных генов и разновидностей. Если общее число обоих свойств превышает 100, мы вызываем соответствующий дорожке метод `configure()`, чтобы установить параметр `bump` в `false` (17–18).

ЧТЕНИЕ КОДА, ПРЕДСТАВЛЕННОГО В ЭТОЙ ГЛАВЕ

`Bio::Graphics` и `BioPerl` написаны на Perl, языке, который обманчиво похож на Си или Java, но имеет множество загадочных особенностей. Если вы не знакомы с языком Perl, то не стоит волноваться. Просматривайте примеры кода, для того чтобы уловить общий смысл логики конструкции. Чтобы помочь вам разобраться с тем, что происходит, предлагаю вашему вниманию краткий обзор наиболее изощренных частей синтаксиса Perl:

`$имя_переменной`

Имя переменной, начинающееся со знака доллара (\$), является скалярной переменной. В такой переменной хранится только одно строковое или числовое значение.

`@имя_переменной`

Имя переменной, начинающееся со знака «эт» (@), обозначает упорядоченный массив. В такой переменной хранится несколько строк или чисел, индексированных по их числовой позиции в массиве. При адресации элемента массива индекс следует помещать в квадратные скобки:

```
$foo[3] = 'третий элемент';
```

Знак \$ стоит перед имени переменной, потому что отдельный элемент представляет собой скалярную величину. При определении подпрограммы или метода специальный массив, названный @_ содержит перечень аргументов, переданных подпрограмме.

`%имя_переменной`

Имя переменной, начинающееся со знака процента (%), обозначает неупорядоченный хэш. В этом хэше содержатся несколько строк или чисел (значения хэша), индексированные строками (ключами хэша). Когда хэшу назначаются пары ключ–значение, используется следующая система записи:

```
%виды = (обезьяна=>'млекопитающее',
лягушка=>'амфибия',
газель=>'млекопитающее');
```

При адресации к элементу хэша ключ помещается в фигурные скобки:

```
print «Обезьяна относится к виду ». %виды{обезьяна};
$объект->метод('арг1','арг2','арг3')
```

Стрелка -> указывает на вызов объектно-ориентированного метода. Объект хранится в скалярной переменной \$объект. Вызываемый метод принимает от нуля до нескольких аргументов.

Если аргументы отсутствуют, скобки можно опустить, что выглядит весьма странно для знатоков Си и Java:

```
$объект->метод;
```

Внутри определения метода объект обычно хранится в скалярной переменной под названием \$self, хотя это вопрос применяемого стиля программирования.

Тем самым включается контроль конфликтов для дорожек, в которых содержится управляемое количество свойств, чтобы накладывающиеся друг на друга свойства перемещались вверх и вниз, не перекрывая друг друга. Для дорожек с большим количеством свойств, контроль конфликтов для которых может привести к их неприемлемой высоте, параметр контроля устанавливается в `false`, чтобы перекрывающиеся свойства накладывались друг на друга.

Установка параметров

К этому моменту я еще не понял, как будут установлены параметры. Я решил, что для выдерживания стиля программирования на BioPerl параметры должны передаваться в виде пар `тег-значение`, в формате `-параметр_имя=>параметр_значение`. К примеру, следующий вызов метода может привести к созданию дорожки со свойством, нарисованной высотой в 10 пикселов, с синим фоновым цветом и с включенным контролем конфликтов (`bump`):

```
$track1 = $panel_object->add_track(\@features,  
    -height => 10,  
    -bgcolor => 'blue',  
    -bump => 1);
```

Позже, с помощью вызова метода `configure()`, можно было бы изменить любой из параметров. В этом примере показано, как выключается контроль конфликтов:

```
$track1->configure(-bump => 0);
```

В конечном счете я расширил диапазон параметров конфигурации дорожки, предоставив возможность в качестве значений параметров использовать кодовые ссылки (типа обратного вызова), но у первой пробы этого модуля это свойство отсутствовало. Оно будет рассмотрено в этой главе чуть позже.

Итак, какие же параметры конфигурации должны поддерживаться объектом дорожки? Я быстро составил небольшой набор стандартных параметров дорожки, самым главным из которых был параметр `glyph`:

```
-glyph => 'glyph_type'
```

Как уже ранее говорилось, я хотел получить возможность поддержки широкого диапазона визуальных представлений свойств (глифов). Параметр `-glyph` был для конечного разработчика способом доступа к этому диапазону представлений. К примеру, `-glyph=>'box'` приведет к формированию изображения свойств в виде простых прямоугольников, `-glyph=>'oval'` приведет к формированию изображения свойств в виде подходящего размера овалов и `-glyph=>'arrow'` приведет к появлению стрелок.

В добавление к параметру `-glyph` в исходную конструкцию были включены другие параметры:

```
-fgcolor
```

цвет переднего плана (строки) свойств, отображенных в дорожке

```
-bgcolor
```

цвет заднего плана (закраски) свойств, отображенных в дорожке

-bump

выключатель контроля конфликтов

-label

выключатель вывода названия над каждым свойством

-description

выключатель вывода описания под каждым свойством

-height

высота отображения каждого свойства

-key

общее обозначение дорожки

-tkcolor

фоновый цвет дорожки

Я осознавал, что у глифов могли бы быть параметры специального назначения, отсутствующие у менее сложных параметров, поэтому мне надо было так спроектировать библиотеку кода, чтобы список параметров, передаваемых методу `add_track()`, был бы расширяемым.

Панель также нуждалась в параметрах для таких составляющих, как желаемая ширина изображения и масштаб перевода пикселов в пары нуклеотидов.

Я составил небольшую подборку параметров, связанных с панелью, которая включала:

-width

ширину изображения в пикселях

-length

длину участка последовательности в парах нуклеотидов

Теперь я мог конкретизировать историю кода, чтобы отобразить специфику каждого вызова `Bio::Graphics`, как показано в примере 12.3.

Пример 12.3. Конкретизированная история использования `BioPerl::Graphics` (псевдокод)

```
1 use Bio::Graphics::Panel;
2 $panel_object = Bio::Graphics::Panel->new(-width => 800,
3                                              -length => 50000);
3 $track1 = $panel_object->add_track(-glyph => 'box',
4                                     -height => 12,
4                                     -bump   => 1,
4                                     -key    => 'Protein-coding Genes');
4 $track2 = $panel_object->add_track(-glyph => 'triangle',
4                                     -height => 6,
4                                     -bump   => 1,
4                                     -key    => 'Sequence Variants');
5 $collection = Bio::SeqFeature::CollectionI->new(@args);
6 $iterator = $collection->get_seq_stream(@query_parameters);
```

```

7 $genes=0; $variations=0;
8 while ($feature = $iterator->next_seq) {
9     if ($feature->method eq 'gene') {
10         $track1->add_feature($feature);
11         $genes++;
12     } elsif ($feature->method eq 'variation') {
13         $track2->add_feature($feature);
14         $variations++;
15     }
16 }
17 $track1->configure(-bump => 0) if $genes > 100;
18 $track2->configure(-bump => 0) if $variations > 100;
19 print $panel_object->png;

```

Выбор классов объектов

Последней основной конструкторской задачей был выбор основных классов объектов, с которыми разработчик мог бы организовать взаимодействие. Исходя из истории кода сначала просматривались два основных класса.

Bio::Graphics::Panel

Объекты этого класса будут представлять диаграмму в целом и будут, как правило, поделены на ряд горизонтальных дорожек. *Bio::Graphics::Panel* будет отвечать за позиционирование каждой дорожки в области ее отображения и за конвертирование координат свойств, выраженных в парах нуклеотидов, в координатах глифов, выраженных в пикселях.

Bio::Graphics::Track

Объекты этого класса будут представлять дорожки, составляющие панель. Дорожки, в первую очередь, будут отвечать за позиционирование и рисование глифов.

А что же насчет самих глифов? Представляется вполне естественным, что глифы должны быть объектами и содержать внутреннюю логику для своего собственного рисования. Все глифы должны получать наследство от объекта *Bio::Graphics::Glyph*, который посвящен в основы рисования. Как только будет вызван метод *add_feature()*, принадлежащий объекту *Track*, за счет вызова конструктора объекта *Glyph* будут созданы новые глифы:

```
$glyph = Bio::Graphics::Glyph->new(-feature=>$feature);
```

Затем, когда объекту дорожки — *Track* потребуется самого себя нарисовать, он задействует метод *draw()*, который будет выглядеть следующим образом:

```

sub draw {
    @glyphs = $self->get_list_of_glyphs();
    for $glyph (@glyphs) {
        $glyph->draw();
    }
    # рисование других элементов дорожки, например ее названия
}

```

Эта подпрограмма начинается с выборки списка объектов `Glyph`, которые были созданы во время работы метода `add_feature()`. Затем для каждого глифа, чтобы он сам себя нарисовал, вызывается его метод `draw()`. В заключение рисуются специфические для данной дорожки элементы, в частности ее название.

Дальнейшее размышление о `Bio::Graphics::Glyph` навело меня на мысль, что эти объекты должны заключать в себе некую интеллектуальную составляющую. Нужно вспомнить, что свойство последовательности должно иметь внутреннюю структуру, с подсвойствами, подподсвойствами и т. д., и каждый из компонентов внутренней структуры должен быть выложен с применением контроля конфликтов, а затем нарисован в соответствии с пользовательскими предпочтениями. Поведение, касающееся раскладки компонентов и их прорисовки, очень похоже на поведение глифов, поэтому мне представилось, что есть смысл дать глифам возможность иметь подглифы в параллель со структурой свойство—подсвойство. Тогда метод `new()` объекта `Glyph` будет выглядеть следующим образом:

```
sub new {
    $self = shift; # получение указателя на свой же объект
    $feature = shift; # получение указателя на свойство
    for $subfeature ($feature->get_SeqFeatures) {
        $subglyph = Bio::Graphics::Glyph->new(-feature=>$subfeature);
        $self->add_subpart($subglyph);
    }
}
```

Для каждого имеющегося в свойстве подсвойства мы создаем новый подглиф и добавляем его к внутреннему списку. Поскольку вызов метода `new()` осуществляется рекурсивно, если подсвойство, в свою очередь, тоже имеет подсвойство, создается другой уровень вложенных глифов.

Для того чтобы глиф верхнего уровня нарисовал себя вместе со всеми своими подглифами, его метод рисования должен выглядеть следующим образом:

```
sub draw {
    @subglyphs = $self->get_subparts();
    for $subglyph (@subglyphs) {
        $subglyph->draw();
    }
    # прорисовка
}
```

В данном фрагменте псевдокода для извлечения всех подглифов, созданных нашим конструктором, вызывается метод `get_subparts()`. Он осуществляет последовательный перебор всех подглифов и вызывает их методы `draw()`. Затем код осуществляет саму прорисовку глифа.

И в этот момент меня поразило то, что псевдокод метода `draw()`, относящегося к объекту `Glyph`, был как две капли воды похож на псевдокод показанного ранее метода `draw()`, относящегося к объекту `Track`. Я понял, что эти два класса можно объединить за счет простого приспособления метода `add_track()` под создание отдельного объекта внутреннего свойства, связанного с дорожкой, и управления этим объектом. Последующие вызовы метода `add_feature()` фактически будут добавлять к свойству подсвойства.

Я опробовал некий тестовый код и пришел к выводу о жизнеспособности этой затеи. В добавок к преимуществам избавления от излишнего кода рисования появилась возможность объединения всего кода, имеющего отношение к передаче и конфигурации параметров дорожки и глифа. Таким образом, дорожки (*tracks*) стали подклассом *Bio::Graphics::Glyph*, названным *Bio::Graphics::Glyph::track*, а метод *add_track()*, относящийся к объекту *Panel*, приобрел следующий (здесь несколько упрощенный) вид:

```
sub add_track {
    my $self = shift;
    my $features = shift;
    my @options = @_;
    my $top_level_feature = Bio::Graphics::Feature->new(-type=>'track');
    my $track_glyph =
        Bio::Graphics::Glyph::track->new(\@options);

    if ($features) {
        $track_glyph->add_feature($_) foreach @$features;
    }
    $self->do_add_track($track_glyph);
    return $track_glyph;
}
```

Чтобы обеспечить соответствие самой первой истории кода, в которой вызывающая программа передает список свойств методу *add_track()*, я дал возможность первому параметру стать списком свойств. В реальном коде я осуществляю во время выполнения проверку типа первого аргумента, чтобы отличить список свойств от первого параметра. Это позволяет вызывающей программе вызвать метод *add_track()* либо в стиле, рассмотренном в первой истории кода:

```
$panel->add_track(\@list_of_features,@options)
```

либо в стиле второй истории кода:

```
$panel->add_track(@options)
```

Затем метод *add_track()* создает новое свойство типа *track*, используя облегченный класс свойства (*feature*), который я написал для *Bio::Graphics* (необходимость в этом возникла из соображений производительности; стандартный объект свойства, имевшийся в *BioPerl*, задействовал слишком много памяти, что приводило к ее снижению). Этот класс передавался конструктору для *Bio::Graphics::Glyph::track*.

Если методу был предоставлен список свойств, он осуществляет последовательный перебор этого списка и вызывает относящийся к глифам дорожки метод *add_feature()*.

В заключение метод *add_track()* добавляет дорожку к внутреннему списку дорожек, который был добавлен к панели, и возвращает глиф дорожки вызывающей программе.

Относящийся к дорожке метод *add_feature()* используется для создания содержащихся в дорожке подглифов. Он вызывается или конструктором глифов, или позже разработчиком, когда он осуществляет вызов *\$track->add_feature()*. По идеи, код должен выглядеть следующим образом:

```

sub add_feature {
    my $self = shift;
    my $feature = shift;
    my $subglyph = Bio::Graphics::Glyph->new(-feature=>$feature);
    $self->add_subpart($subglyph);
}

```

Здесь я показываю, что конструктор для `Bio::Graphics::Glyph` вызывается в жестко запрограммированном виде, но на практике мы будем иметь дело с множеством глифов различного типа, поэтому выбор создаваемого подкласса `Bio::Graphics::Glyph` должен быть сделан во время выполнения программы, на основе параметров, предоставленных пользователем. В следующем разделе будет описано решение, к которому я пришел в результате реализации этого замысла.

Обработка параметров

Следующим шагом в процессе конструирования стало определение порядка динамического создания глифов. Этот порядок был неотъемлемой частью общей задачи по обработке параметров настраиваемой пользователем конфигурации. В соответствии с историей кода я хотел, чтобы параметры были определены при создании дорожки:

```
$panel->add_track(-glyph => 'arrow',
                     -fgcolor => 'blue',
                     -height => 22)
```

В этом примере у панели запрашивается создание дорожки, содержащей глифы в виде стрелок — `arrow`, чей передний план окрашен в синий цвет, а высота составляет 22 пикселя. В соответствии с решением, принятым в предыдущем разделе, метод `add_track()` создаст жестко заданный глиф дорожки типа `Bio::Graphics::Glyph::track`, и передаст эти параметры его конструктору `new()`. Позже, при вызове метода `add_feature()`, относящегося к глифу дорожки, им будет создан новый подглиф для каждого отображаемого свойства.

Тем не менее оставались нерешенными три вопроса.

1. Как относящийся к глифу дорожки метод `add_feature()` определит, какой тип глифа нужно создавать?

На основе предпочтений пользователя мы хотим для отображения разных типов свойств создавать разные типы глифов. Так, в предыдущем примере пользователь захотел заполнить дорожку серией глифов в виде стрелки — `arrow`, основываясь на значении параметра `-glyph`. В предыдущем разделе псевдокод для `$track->add_feature()` имел жестко заданный вызов `Bio::Graphics::Glyph->new()`, но в коде программного продукта мы хотим осуществить динамический выбор соответствующего подкласса глифа — например `Bio::Graphics::Glyph::arrow`.

2. Как эти подглифы узнают, какого типа подподглифы нужно создавать?

Вспомним о том, что свойства должны содержать подсвойства, а каждое подсвойство должно быть представлено подглифом, который является частью

основного глифа. В предыдущем примере глиф дорожки, основываясь на значении параметра `-glyph`, сначала создает серию глифов в виде стрелок. Затем глифы-стрелки несут ответственность за создание любых необходимых им подглифов; эти подглифы несут ответственность за создание своих подглифов и т. д. Но как глиф-стрелка примет решение о том, какого типа подглиф создавать?

3. Как только что созданному глифу передаются другие параметры?

Например, какой объект в приведенном примере отслеживает значения параметров `-fgcolor` и `-height`?

Поскольку выбор типа глифа является особой разновидностью обработки параметров конфигурации, я решил сначала справиться с решением именно этой проблемы. Первое, что мне пришло в голову, было то, что каждый глиф должен нести ответственность за управление своими параметрами, но энтузиазм в отношении этой идеи угас довольно быстро. Поскольку дорожка могла содержать тысячи глифов, было бы крайне неэффективно для каждого из них содержать полную копию его конфигурации. Я также подумал о хранении параметров в объекте `Panel`, но почувствовал, что этого тоже не следует делать, поскольку у панели были свои собственные параметры, отличающиеся от параметров дорожки.

Придуманное мною решение заключалось в создании для глифов ряда «производителей» — `factories`, относящихся к типу `Bio::Graphics::Glyph::Factory`. При каждом создании дорожки `Panel` создает соответствующий производитель, инициализированный параметрами, желательными для вызывающей программы. Каждый имеющийся на дорожке глиф и подглиф содержит ссылку на производитель и осуществляет вызовы производителя для получения его параметров. Следовательно, если у панели есть четыре дорожки, значит, есть и четыре объекта `Bio::Graphics::Glyph::Factory`.

Поскольку я пришел к идеи производителя, решение задачи создания соответствующих типов глифов и подглифов существенно упростилось. Производитель сохраняет глиф, выбранный пользователем (например стрелку `-arrow`), со всеми его другими параметрами. Производитель содержит метод, названный `make_glyph()`, создающий необходимые глифы и подглифы, используя сохраненные параметры для решения, подкласс какого глифа нужно использовать.

В этом варианте подразумевается, что все глифы, содержащиеся внутри дорожки, совместно используют один и тот же класс. Иными словами, если конкретное свойство содержит три вложенных уровня подсвойств и пользователь выбрал для свойств дорожки глиф стрелки `-arrow`, то каждый глиф-стрелка содержит подглифы-стрелки, которые, в свою очередь, содержат подподглифы-стрелки. Хотя это выглядит как существенные ограничения, но на самом деле имеет некоторый смысл. Как правило, глиф и его подразделения работают вместе, и то, что все они выполнены в едином подклассе, позволяет хранить весь соответствующий код в одном месте. Кроме того, глифы могут обойти это ограничение, подменив свои конструкторы `new()`, чтобы создать подглифы выбранного типа.

В окончательной версии класса `Bio::Graphics::Glyph::Factory` содержится всего несколько методов.

Конструктор

Конструктор создает нового производителя — `factory`:

```
$factory = Bio::Graphics::Glyph::Factory->new(-options=> \%options,
                                              -panel => $panel);
```

Во время работы конструктор принимает список параметров, переданных ему методом панели `add_track()`, и сохраняет их внутри. Производитель также может хранить копию панели. Я добавил эту возможность исходя из того, что производитель должен предоставлять информацию о панели, в частности о ее масштабе.

В действительности параметры передаются в виде ссылки на хэш (используемый в Perl словарь, состоящий из пар имя–значение). Принадлежащий панели метод `add_track()` несет небольшую нагрузку по превращению переданного ему списка пар `-option=>$value` в хэш для передачи методу производителя `new()`.

Метод option()

Получив имя параметра, производитель ищет и возвращает его значение:

```
$option_value = $factory->option('option_name')
```

Если параметры с таким именем не установлены, `option()` смотрит, нет ли значения по умолчанию, и возвращает это значение.

Метод make_glyph()

Получив список свойств, производитель создает список глифов соответствующего класса:

```
@glyphs = $factory->make_glyph($свойство1,$ свойство2,$ свойство3...)
```

Теперь мы взглянем на упрощенную версию кода `Bio::Graphics::Glyph::Factory`:

```
1 package Bio::Graphics::Glyph::Factory;
```

```
2 use strict;
```

```
3 my %GENERIC_OPTIONS = (
4     bgcolor => 'turquoise',
5     fgcolor => 'black',
6     fontcolor => 'black',
7     font2color => 'turquoise',
8     height => 8,
9     font => 'gdSmallFont',
10    glyph => 'generic',
11 );
12 sub new {
13     my $class = shift;
14     my %args = @_;
15     my %options = $args{-options}; # параметры в виде ссылки на хэш
16     my $panel = $args{-panel};
17     return bless {
18         options => %options,
19         panel => $panel,
```

```
20         }.$class;
21 }

22 sub option {
23     my $self = shift;
24     my $option_name = shift;
25     $option_name = lc $option_name; # все параметры переводятся в нижний
# регистр
26     if (exists $self->{options}{$option_name}) {
27         return $self->{options}{$option_name};
28     } else {
29         return $GENERIC_OPTIONS{$option_name};
30     }
31 }

32 sub make_glyph {
33     my $self = shift;
34     my @result;

35     my $glyph_type = $self->option('glyph');
36     my $glyph_class = 'Bio::Graphics::Glyph::' . $glyph_type;
37     eval("require $glyph_class") unless $glyph_class->can('new');

38     for my $feature (@_) {
39         my $glyph = $glyph_class->new(-feature => $f,
40                                         -factory => $self);
41         push @result, $glyph;
42     }
43     return @result;
44 }
45 1;
```

Я начал с объявления пакета и включения строгой проверки типов (строки 1 и 2).

Затем определил характерный для пакета хэш, содержащий некоторые групповые параметры глифа, чтобы использовать их в качестве аварийных значений по умолчанию. Среди этих параметров – используемые по умолчанию цвет фона, высота и шрифт (строки 3–11).

Конструктор new() считывает свои аргументы из @_ (списка аргументов подпрограммы, используемого в Perl) в хэш, названный %args. Затем он ищет два поименованных аргумента, -options и -panel, и сохраняет их во внутреннем бе-зымянном хэше под ключами options и panel, создает объект, используя функцию Perl bless, и возвращает его (строки 12–21).

Определение метода option() занимает строки 22–31. Ячитываю объект производителя и запрашиваю имя параметра из списка аргументов подпрограммы. Затем вызываю встроенную функцию lc(), чтобы перевести имя параметра в нижний регистр и защитить поведение метода от разработчиков, забывших, как именно назван параметр: -height или -Height. В хэше параметров, который я создал при работе метода new(), я определяю наличие ключей с таким же именем, и если они присутствуют, возвращаю соответствующие им значения. В противном случае я использую имя параметра для индексации внутри %GENERIC_OPTIONS и возвращаю соответствующее значение. Если соответствующий

ключ отсутствует как в хэше параметров, так и в %GENERIC_OPTIONS, я заканчиваю тем, что возвращаю неопределенное значение.

Метод `make_glyph()` (строки 32–44) демонстрирует возможности Perl по динамической загрузке модуля во время исполнения программы. Сначала я ищу желаемый тип глифа, используя `option()` для поиска значения параметра `glyph`. Заметьте, что пара ключ–значение `glyph=>'generic'` определена в %GENERIC_OPTIONS; это означает, что если программист проигнорирует запрос определенного типа глифа, `option()` вернет `generic`.

Теперь, если нужно, я загружаю класс запрошенного глифа. По соглашению все подклассы `Bio::Graphics::Glyph` называются `Bio::Graphics::Glyph::имя` подкласса. Глиф `generic` располагает классом Perl `Bio::Graphics::Glyph::generic`, глиф `arrow` находится в `Bio::Graphics::Glyph::arrow` и т. д. Для создания абсолютно правильного имени класса я воспользовался операцией объединения строк (`..`). Затем я скомпилировал и загрузил этот класс в память, используя `require $glyph_class`. Вызов `require` заключен в строку и передан компилятору Perl с помощью метода `eval()`. Это сделано, чтобы удержать Perl от попытки вызвать `require()` во время компиляции определения `Factory`. Чтобы избежать ненужных перекомпиляций, я загружаю класс, только если обнаружу, что конструктор `new()` еще не существует, что свидетельствует о том, что класс еще не загружен.

Я осуществляю последовательный перебор каждого параметра, переданного в массив аргументов подпрограммы `@_`, вызывая конструктор `new()`, принадлежащий классу выбранного глифа. Каждый заново созданный глиф помещается в массив, который затем возвращается вызывающей программе.

В последней строке модуля стоит 1, которой, большей частью по историческим мотивам, завершаются все модули Perl.

Заметьте, что конструктор глифов теперь расширен, и каждый конструируемый глиф использует два поименованных аргумента: свойство и объект производителя. Благодаря передаче копии производителя каждый глиф может добраться до относящихся к нему параметров. Посмотрите на выборки двух относящихся к этому методов из `Bio::Graphics::Glyph`:

`factory()`

Возвращает объект производителя, переданный глифу при его конструировании:

```
sub factory {
    my $self = shift;
    return $self->{factory};
}
option( )
```

Это проходной метод для получения значения названного параметра:

```
sub option {
    my $self = shift;
    my ($option_name) = @_;
    return $self->factory->option($option_name);
}
```

Глиф вызывает `factory()`, чтобы получить своего производителя и тут же вызвать метод производителя `option()` для получения значения параметра, определенного в списке аргументов подпрограммы.

Пример кода

Чтобы собрать все вместе, в примере 12.4 представлена простая иллюстрация Bio::Graphics в действии. Выводимая информация показана на рис. 12.2.

Пример 12.4. Сценарий, использующий Bio::Graphics

```
1 #!/usr/bin/perl
2 use strict;
3 use Bio::Graphics;
4 use Bio::SeqFeature::Generic;
5 my $bsg = 'Bio::SeqFeature::Generic';
6 my $span      = $bsg->new(-start=>1,-end=>1000);
7 my $test1_feat = $bsg->new(-start=>300,-end=>700,
8                            -display_name=>'Test Feature',
9                            -source_tag=>'This is only a test');
10 my $test2_feat = $bsg->new(-start=>650,-end=>800,
11                            -display_name=>'Test Feature 2');
12 my $panel= Bio::Graphics::Panel->new(-width=>600, -length=>$span->length,
13                                         -pad_left=>12,-pad_right=>12);
14 $panel->add_track($span,-glyph=>'arrow',-double=>1,-tick=>2);
15 $panel->add_track([$test1_feat,$test2_feat],
16                     -glyph => 'box',
17                     -bgcolor => 'orange',
18                     -font2color => 'red',
19                     -height => 20,
20                     -label => 1,
21                     -description => 1,
22 );
23 print $panel->png;
```

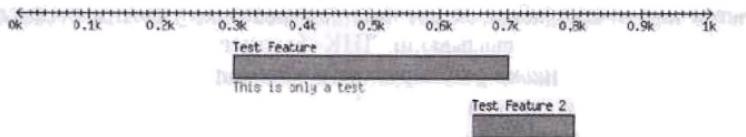


Рис. 12.2. Изображение, выведенное в результате работы примера 12.4

Мы загружаем библиотеку Bio::Graphics и один из стандартных классов Bio-Perl — Bio::SeqFeature::Generic из состава Bio::SeqFeatureI (строки 3–4). Чтобы избежать многократного ввода полного имени класса свойств, мы сохраняем его в переменной (строка 5).

Затем мы создаем три объекта `Bio::SeqFeature::Generic`. Отображение одного из них начинается с позиции 1 и заканчивается на позиции 1000 и будет использовано для рисования дорожки, содержащей шкалу для изображения (строка 6). Два других станут свойствами на второй дорожке (строки 7–11).

Мы создаем панель, передавая ей параметры, определяющие ее ширину в пикселях и длину в парах нуклеотидов, и дополнительное незаполненное пространство, чтобы создать бордюр изображения слева и справа (строки 12–13).

Затем мы создаем дорожку для шкалы изображения (строка 14). Она состоит из единственного свойства, содержащегося в переменной `$span`, и параметров, в которых выбран глиф стрелки — `arrow`, эта стрелка сделана двуглавой (`-double=>1`), и на ней пропечатаны как крупные, так и мелкие градуировочные метки (`-tick=>2`).

Теперь настало время создать дорожку для двух свойств, `$test1_feat` и `$test2_feat`. Мы добавляем вторую дорожку, указывая на этот раз параметр для использования глифа прямоугольника — `box`, имеющего оранжевый цвет фона, красный цвет описания и высоту 20 пикселов. Мы также выборочно включаем вывод на экран имени свойства и описания (строки 15–22).

Последний шаг заключается в вызове метода объекта панели `png()` для конвертации изображения в графику формата PNG, и мы печатаем график на стандартном устройстве вывода, где он может быть сохранен в файле или переправлен программе обслуживания графического дисплея (строка 23).

Динамические параметры

Исходная конструкция `Bio::Graphics::Glyph::Factory` была основана на идее простых статических значений параметров. Но как только я приступил к работе над первой версией `Bio::Graphics`, то сразу понял, что куда удобнее будет предоставить разработчику возможность динамического вычисления некоторых параметров.

Например, разбросанные по геному участки ДНК, где присоединяется регулирующий белок. Когда этот белок присоединяется к определенному месту ДНК (процесс, называемый «связывание»), близлежащий ген, как правило, включается или выключается. Некоторые связываемые участки бывают сильнее, а другие слабее, и сила связывания ДНК–белок часто играет очень важную роль в понимании механизма регулирующего взаимодействия.

Для создания дорожки, показывающей позиции и относительную силу связывания свойств участка ДНК–белок, разработчику может потребоваться показать ряд прямоугольников. Размах свойства будет показан начальной и конечной позицией прямоугольника, а сила связывания — его фоновым (внутренним) цветом: белым — для слабого связывания, розовым — для среднего по силе и красным — для сильного связывания. Используя исходную конструкцию, разработчик может определить фоновый цвет для всех свойств дорожки:

```
@features = get_features_somewhat();
$panel->add_track(\@features,
```

```
-glyph => 'generic'.
-bgcolor => 'pink');
```

Но она не предлагает способа установки цвета от свойства к свойству.

Когда обнаружилось это ограничение, я вернулся назад и расширил API, позволяя значениям параметров быть ссылками на блоки CODE, представляющие собой безымянные процедуры, которые Perl-программисты могут определять множеством способов и использовать примерно так же, как в языке Си используются указатели. Вот как теперь выглядит переделанный вызов `add_track()`, в котором используются преимущества этого свойства:

```
$panel->add_track(\@features,
  -glyph => 'box',
  -bgcolor => sub {
    my $feature = shift;
    my $score = $feature->score;
    return 'white' if $score < 0.25;
    return 'pink' if $score < 0.75;
    return 'red';
});
```

Он работает следующим образом: значение параметра `-bgcolor` является анонимной ссылкой на блок CODE, которая создана с помощью ключевого слова `sub` без использования имени подпрограммы. Код вызывается во время выполнения программы, когда глифу требуется получить значение своего параметра `bgcolor`. Подпрограмма в своем массиве аргументов получает соответствующее свойство и вызывает свой метод `score()` для получения силы связывания с участком. Предполагая, что сила связывания участка выражается числом с плавающей точкой в диапазоне от 0 до 1,0, я возвращаю значение параметра `white` (белый), если оценка этой силы меньше, чем 0,25, а значение `pink` (розовый) — если оценка больше, чем 0,25, но меньше, чем 0,75, и значение `red` (красный), если она больше, чем 0,75. На рис. 12.3 показано, как это может выглядеть.



Рис. 12.3. Раскраска фона в соответствии с динамически изменяемыми значениями

В конечном счете, я сделал возможным использование обратного вызова для каждого параметра, передаваемого методу `add_track()`, включая сам параметр `-glyph`. Тем самым конечному пользователю предоставилась удивительная гибкость в настройке библиотеки под свои нужды и ее расширении. К примеру, значительно упростилось «семантическое изменение масштаба» или изменение внешнего вида дорожек в зависимости от размера отображаемого участка.

Следующий обратный вызов выключает контроль конфликтов, когда участок занимает более 50 000 пар нуклеотидов:

```
-bump => sub {
  my ($feature,$option_name,$glyph) = @_;
  # get all args
```

```
    return $glyph->panel->length < 50_000;
}
```

Посмотрим теперь на упрощенную версию пересмотренного кода обработки параметров. Сначала я изменил Bio::Graphics::Glyph::Factory:

```
# Код из Bio::Graphics::Glyph::Factory
sub option {
    my $self = shift;
    my ($glyph,$option_name) = @_;
    $option_name = lc $option_name; # all options are lowercase
    my $value:
    if (exists $self->{options}{$option_name}) {
        $value = $self->{options}{$option_name};
    } else {
        $value = $GENERIC_OPTIONS{$option_name};
    }

    return $value unless ref $value eq 'CODE';

    my $feature = $glyph->feature;
    my $eval = eval {$value->($feature,$option_name,$glyph)};
    warn "Error while evaluating \"$option_name\" option for glyph $glyph.
feature $feature: ".$@."\n"
    if $@:

    return defined $eval && $eval eq '*default' ?
        $GENERIC_OPTIONS{$option_name}
        : $eval;
}
```

Теперь метод принимает не один, а два аргумента. В качестве первого выступает текущий глиф, а в качестве второго, как и раньше, имя параметра. Еще раз вспомним, что производитель (factory) сначала заглядывает в свой хэш с параметрами, относящимися к дорожке, а затем в хэш параметров по умолчанию (%GENERIC_OPTIONS), если параметр не был назван в конфигурации дорожки.

Но теперь, после извлечения значения параметра, вступает в силу дополнительная логика. Чтобы найти тип данных в содержимом \$value, я вызываю имеющуюся в Perl функцию ref(). Если ссылка на код имеется, ref() возвращает строку CODE. Если строка CODE не получена, я, как и раньше, просто возвращаю значение. В противном случае я получаю соответствующее свойство за счет вызова имеющегося в объекте глифа метода feature(), а затем вызываю ссылку на код, используя имеющийся в Perl синтаксис вызова ссылки на безымянный код:

```
$value->($feature,$option_name,$glyph)
```

Первый аргумент, передаваемый обратному вызову, является свойством, второй — именем параметра, а третий — самим глифом.

Поскольку обратный вызов может привести к ошибке времени выполнения, я защитился от возможности ее появления, заключив весь вызов в блок eval{}. Если при обратном вызове произойдет неисправимая ошибка, этот блок вернет неопределенное значение и поместит отправленное Perl диагностическое

сообщение об ошибке в специальной скалярной переменной \$@. После осуществления обратного вызова я проверяю переменную \$@ на заполнение, и если в ней что-то есть, вывожу предупреждение.

Последним шагом является возврат значения, полученного из обратного вызова. Я посчитал, что для обратного вызова будет полезным наличие возможности указать на то, что для названного параметра нужно воспользоваться значением по умолчанию. В последней строке кода выполняется простая проверка на то, что обратный вызов вернул строку *default*, и если так оно и есть, то возвращается значение из хэша значений по умолчанию.

Чтобы согласовать это изменение с имеющимся в производителе методом option(), мне пришлось внести соответствующие изменения в Bio::Graphics::Glyph->option():

```
# Кол из Bio::Graphics::Glyph
sub option {
    my $self = shift;
    my ($option_name) = @_;
    return $self->factory->option($self,$option_name);
}
```

Во время работы с обратными вызовами я все больше понимал, насколько они полезны. К примеру, я понял, что они очень хорошо справляются с семантическим изменением масштаба. Генные глифы рисуют подробное представление внутренней структуры кодирующих белок генов, которые хорошо выглядят при большом увеличении, но не работают при просмотре очень больших участков, где детали становятся настолько маленькими, что их уже невозможно различить. Между тем к параметру -glyph можно применить обратный вызов, чтобы в динамическом режиме выбрать глиф в виде простого прямоугольника – box, а не глиф gene, в том случае когда ген составляет меньше 5% от размера отображаемого участка:

```
$panel->add_track(
    -glyph => sub {
        my ($feature,$panel) = @_;
        return 'box' if $feature->length/$panel->length < 0.05;
        return 'gene';
    },
    -height => 12,
    -font2color => 'red',
    -label_transcripts => 1,
    -label => 1,
    -description => 1,
);
```

Заметьте, что для параметра -glyph аргументы обратного вызова отличаются от аргументов для других параметров, поскольку это значение требуется получить до создания глифа. Вместо передачи свойства, имени параметра и глифа, обратный вызов передает свойство и объект панели. Получилось так, что свойство обратного вызова стало одним из самых популярных свойств Bio::Graphics.

По прошествии времени я не скучая добавил обратные вызовы к другим частям API, включая ту, которая обрабатывала параметры, передаваемую

конструктору панели, и код, который решает, в каком порядке проводить сортировку свойств сверху вниз.

Пользователи нашли применение обратным вызовам в самых неожиданных для меня различных случаях. Приведу типичный пример. Я предоставил пользователям способ определять обратный вызов, позволяющий выполнять рисунки на панели напрямую, после того как будут нарисованы линии сетки, но до того как будут нарисованы глифы. Через несколько лет предпримчивый гений биолог понял, как воспользоваться этим свойством, чтобы создать диаграммы, которые сравнивают гены представителей биологического вида, чьи хромосомы подверглись структурным изменениям относительно друг друга. Обратный вызов линий сетки рисует раскрашенные многоугольники, соединяющие свойства одной хромосомы с соответствующими свойствами другой (рис. 12.4).

Но в истории Bio::Graphics::Factory есть и свои темные стороны. В порыве начального энтузиазма я добавил методам получения и установки параметров множество других свойств, которые я не показывал в приведенных здесь примерах. Одно из таких свойств заключалось в возможности инициализации производителя, используя каскадную таблицу стилей по примеру веб-приложений. Другое свойство предоставляло подробную информацию каждому обратному вызову, относящуюся к связям текущего глифа с другими глифами дорожки или с глифом самого верхнего уровня. На практике эти свойства никогда не использовались и теперь зависли мертвым кодом.

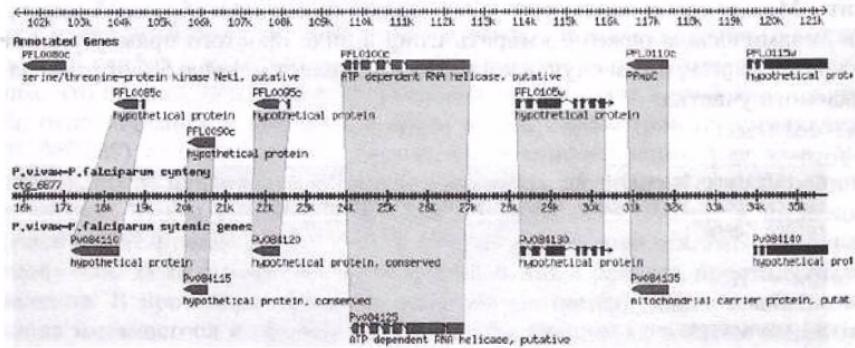


Рис. 12.4. Разумное использование обратных вызовов Bio::Graphics позволило сравнивать две хромосомы взаимосвязанных свойств

Расширение Bio::Graphics

Теперь рассмотрим некоторые расширения библиотеки Bio::Graphics, которые были добавлены после ее первоначального выпуска. Это иллюстрирует развитие кода в ответ на результаты исследований пользовательского опроса.

Поддержка веб-разработчиков

Одной из целей разработки Bio::Graphics была поддержка интерактивных, доступных для просмотра на браузерах представлений геномов с использованием веб-приложений. Моя основная идея в этом плане состояла в том, чтобы CGI-скрипт занимался обработкой заполнения формы, показывающей геном браузеру, и отображаемого участка. Скрипт должен был подключаться к базе данных, обрабатывать пользовательские запросы, разыскивать интересующий участок или участки, извлекать свойства из соответствующего участка и передавать их библиотеке Bio::Graphics, которая будет формировать изображение, а CGI-скрипт будет включать эти данные в отображаемый -тег.

Одним из упущений в этой картине была возможность генерировать карту для генерируемого изображения. Карта изображения необходима для поддержки пользовательской возможности щелкнуть на глифе и получить о нем более подробную информацию. Карта изображения позволяет также создавать подсказки, появляющиеся при прохождении указателя мыши над глифом, и выполнять такие задачи динамического HTML, как заполнение контекстного меню, появляющегося при щелчке пользователя правой кнопкой мыши на глифе.

Чтобы осуществить поддержку генерации карты изображения, исходная версия Bio::Graphics имела единственный метод, названный boxes(). Он возвращал массив, содержащий прямоугольники, охватывающие глифы, свойства, связанные с каждым глифом, и сами глиф-объекты. Для генерации карты изображения разработчикам нужно было перебрать этот массив и вручную сгенерировать HTML-код для карты изображения.

К сожалению, все оказалось не так легко, как я надеялся, судя по количеству полученных мной пользовательских запросов по поддержке продукта. Поэтому после ряда экспериментов по написанию собственного браузера генома на основе Bio::Graphics я добавил метод image_and_map() к Bio::Graphics::Panel. Вот как выглядит фрагмент кода, иллюстрирующий использование этого метода:

```
$panel = Bio::Graphics::Panel->new(@args);
$panel->add_track(@args);
$panel->add_track(@args);
...
($url,$map,$mapname) = $panel->image_and_map(
    -root => '/var/www/html',
    -url  => '/images',
    -link => sub {
        my $feature = shift;
        my $name = $feature->display_name;
        return "http://www.google.com/search?q=$name";
    }
);
print "<H2>Мой геном</H2>";
print "<IMG SRC='$url' USEMAP='#$mapname' BORDER='0' />";
print $map;
```

Настройка панели и добавление дорожек происходит так же, как и раньше. Затем мы вызываем image_and_map() с тремя параметрами-значение. Аргу-

мент `-root` дает информацию о физическом размещении корневого каталога документов веб-сервера — места, где начинается древо HTML-файлов. Аргумент `-url` указывает, где относительно корневого каталога документов должны храниться сгенерированные Bio::Graphics изображения. Аргумент `-link` является обратным вызовом, осуществляемым Bio::Graphics для прикрепления реагирующей на щелчок мыши ссылки на глиф. В данном случае мы восстанавливаем объект свойства из списка аргументов функции обратного вызова, получаем осмысленное с человеческой точки зрения имя свойства, вызывая метод `display_name()`, и генерируем ссылку на поисковый сервер Google. Некоторые другие параметры `image_and_map()` могут быть использованы для пользовательской настройки и расширения получаемой карты изображения.

Метод генерирует изображение и сохраняет его в файловой системе, в том месте, на которое указывают аргументы `-root` и `-url`. — в данном случае это `/var/www/html/images`. Затем он возвращает результирующий список из трех элементов, содержащий URL для сгенерированного изображения, HTML для карты изображения и имя карты изображения для его использования в теге ``. Затем для использования изображения и его карты мы просто отображаем соответствующий фрагмент HTML.

До настоящего времени существовало два браузера геномов, выполненных по веб-технологиям на основе Bio::Graphics. Один из них, созданный мной и названный GBrowse (<http://www.gmod.org/gbrowse>), сейчас широко используется для отображения большого числа геномов в диапазоне от бактерии до человека. Несмотря на то что он был написан в 2002 году, еще до изобретения асинхронного обновления страниц на основе Ajax-технологии, пользователь перемещается по геному, щелкая на кнопках стрелок и ожидая обновления экрана. Новый браузер, который еще находится в стадии прототипа, Ajax Generic Genome Browser (<http://genome.biowiki.org/gbrowse>), обеспечивает отображению генома функциональность в стиле системы Google Maps. Для перехода образ просто захватывается и плавно перемещается.

Поддержка изображений типографского качества

Другим первичным требованием была поддержка нескольких графических форматов. Чтобы его удовлетворить, я приспособил Bio::Graphics под использование библиотеки Perl GD для низкоуровневых вызовов графики. Эта библиотека, основанная на разработке Тома Бьютэлла (Tom Boutell) libgd (<http://www.libgd.org>), генерирует пиксельные изображения в различных форматах, включая PNG, JPEG и GIF.

Внутри объекта панели — Panel создается и обслуживается графический объект GD, который передается каждой из относящихся к его дорожкам процедур `draw()`. Дорожка в свою очередь передает GD-объект своим глифам, а глифы — своим подглифам.

Метод Bio::Graphics::Panel->png() является простым переходом к GD-методу `png()`:

```
# Код из Bio::Graphics::Panel
sub png {
    my $self = shift;
    my $gd = $self->gd;
    return $gd->png;
}
```

Методы jpeg() и gif() аналогичны этому. У разработчика также есть вариант восстановления необработанного GD-объекта и вызова его метода png():

```
$gd = $panel->gd;
print $gd->png;
```

Преимуществом доступности для общего интерфейса внутреннего GD-объекта заключается в том, что разработчик может выполнять с этим объектом какие-то дополнительные действия, например, встраивать его в большее по размеру изображение или воздействовать на его цвета.

Одним из следствий моего выбора в пользу GD было то, что библиотека Bio::Graphics изначально была ограничена в генерации пиксельных изображений. Эта проблема была решена Тоддом Хэррисом (Todd Harris), когда он написал на Perl модуль GD::SVG (<http://toddot.net/projects/GD-SVG>). Этот модуль имел такой же API, как и GD, но генерировал масштабируемые векторные изображения – Scalable Vector Graphics (SVG), которые могли распечатываться в высоком разрешении без потери детализации и обрабатываться в различных графических приложениях вроде Adobe Illustrator.

После того как я добавил поддержку GD::SVG в Bio::Graphics, стало возможным создавать SVG-изображения простой передачей аргумента -image_class конструктору панели:

```
$panel = Bio::Graphics::Panel->new(-length=>1000,
                                      -width=>600,
                                      -image_class => 'GD::SVG'
                                     );
$panel->add_track.... и т. д. ...
print $panel->gd->svg;
```

Единственное внутреннее изменение, которое мне пришлось внести в Bio::Graphics, заключалось в обработке параметра -image_class и загрузке указанной библиотеки обработки изображений. Это позволило в будущем иметь совместимость с новыми, придерживающимися GD-соглашений, библиотеками. Например, если кто-то напишет модуль GD::PDF, генерирующий графические файлы PDF-формата, Bio::Graphics сможет к нему приспособиться.

Добавление новых глифов

Ко времени своего первого издания Bio::Graphics поддерживала около десятка простых глифов, включая прямоугольники, овалы, стрелки, глифы генов и глифы, рисовавшие белковые и ДНК-последовательности. Каждый из этих глифов располагал несколькими параметрами конфигурации, которые вели к огромному количеству возможных отображений. Но их число все же было конечным, в то время как число типов свойств генома потенциально неисчерпаемо. К нашему

всеобщему удовольствию добавление новых глифов является относительно простой задачей, и со временем мною и другими разработчиками BioPerl к Bio::Graphics было добавлено множество новых глифов. На данный момент существует около 70 разновидностей глифов, чей диапазон простирается от причудливых (вроде звезды Давида) до замысловатых (трехкомпонентной диаграммы для сравнения повторяемости вариантов последовательности в нескольких экземплярах).

Возможность простого расширения существующих глифов для создания новых является весьма ценным свойством. Я проиллюстрирую это в примере 12.5, показав вам, как создается новый глиф под названием hourglass (песочные часы).

Пример 12.5. Глиф песочных часов

```

1 package Bio::Graphics::Glyph::hourglass;
2 use strict;
3 use base 'Bio::Graphics::Glyph::box';
4 sub draw_component {
5     my $self = shift;
6     my ($gd,$dx,$dy) = @_;
7     my ($left,$top,$right,$bottom) = $self->bounds($dx,$dy);
8     # рисование многоугольника песочных часов
9     my $poly = GD::Polygon->new;
10    $poly->addPt($left,$top);
11    $poly->addPt($right,$bottom);
12    $poly->addPt($right,$top);
13    $poly->addPt($left,$bottom);
14    $poly->addPt($left,$top);
15    $gd->filledPolygon($poly,$self->bcolor);
16    $gd->polygon($poly,$self->fcolor);
17 }
18 1:
```

Глиф генерирует песочные часы (рис. 12.5). Вначале определяется имя пакета, которое по соглашению должно начинаться с Bio::Graphics::Glyph:: (строка 1). Затем объявляется, что глиф является наследником Bio::Graphics:: Glyph::box, являющегося простым глифом, рисующим прямоугольник (строка 3).

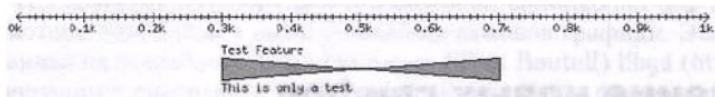


Рис. 12.5. Глиф песочных часов, скрученная версия стандартного прямоугольного глифа

Затем в глифе подменяется унаследованный метод draw_component() (строки 4–17). Этот метод вызывается методом draw() объекта Bio::Graphics::Glyph после настройки среды рисования. Метод получает GD-объект наряду с горизон-

тальными и вертикальными координатами, указывающими позицию глифа относительно того глифа, который его окружает. Мы передаем относительные координаты унаследованному методу `bounds()` для превращения их в абсолютные координаты прямоугольника, включающего глиф (строка 7).

Теперь мы рисуем глиф, создавая многоугольник при помощи имеющейся в GD библиотеки многоугольника и добавки вершин, соответствующих верхнему левому, нижнему правому, верхнему правому и нижнему левому углам песочных часов (строки 9–14). Затем мы передаем объект многоугольника сначала методу `filledPolygon()` GD-объекта, чтобы нарисовать однородное заполнение многоугольника (строка 15), а затем методу `polygon()` GD-объекта, чтобы нарисовать контур песочных часов (строка 16). Обратите внимание на то, как используются унаследованные методы `bgcolor()` и `fgcolor()`, чтобы получить соответствующие цвета для заполнения и контура.

Этим демонстрируется простота добавления к Bio::Graphics новых глифов. Во многих случаях можно создать новый глиф за счет наследования от существующего глифа, который делает почти все, что нужно, с последующей модификацией одного или двух методов, чтобы настроить их под запросы потребителя.

Заключение и извлеченные уроки

Конструирование программного обеспечения, предназначенного для использования другими разработчиками, очень непростая задача. Оно должно быть простым и понятным в использовании, поскольку разработчики также нетерпеливы, как и все остальные; но не может быть настолько примитивным, чтобы это негативно отражалось на функциональности. В идеале библиотека кода должна быть готова к немедленному применению самыми неискушенными разработчиками, легко подстраиваться под нужды более опытных разработчиков и готова к расширению силами экспертов.

Я полагаю, что Bio::Graphics вполне попадает под это определение. Разработчики, не знакомые с BioPerl, могут немедленно приступить к работе, создавая простые сценарии, использующие хорошо знакомые объекты BioPerl, такие как Bio::SeqFeature::Generic. Более опытные разработчики могут настроить выходную информацию библиотеки за счет написания обратных вызовов, а самые опытные — расширить библиотеку за счет собственных глифов.

Bio::Graphics также демонстрирует мощь стандартных интерфейсов. Поскольку эта библиотека была сконструирована для визуализации любого объекта, который следует имеющемуся в BioPerl интерфейсу Bio::SeqFeatureI, она будет работать рука об руку с любыми имеющимися в BioPerl модулями доступа к данным последовательности. Bio::Graphics может генерировать диаграммы из введенных вручную свойств последовательности так же просто, как и отображать свойства, считанные из обычного текстового файла, извлеченные с помощью запроса к базе данных или сгенерированные веб-службой и переданные по сети.

Но в модуле имеется также и ряд недостатков, и если сейчас мне придется бы создавать его заново, то некоторые вещи я сделал бы по-другому.

Основная проблема связана со способом генерации подглифов. В текущей реализации, если вы назначаете глиф свойству и это свойство имеет подсвойства, подглифы будут того же типа, что и глиф высшего уровня.

В этом прослеживаются два изъяна. Во-первых, нужно использовать механизм подклассов, чтобы создать составные глифы, в которых подглифы многократно используют код из ранее определенного класса, а родительский глиф представляет собой нечто новое. Во-вторых, методы глифа должны всегда знать, какой уровень свойства они отображают в данный момент. Например, чтобы создать глиф, в котором верхний уровень представлен пунктирным восьмиугольником, а подсвойства представлены в виде прямоугольников, процедура `draw_component()` должна обеспечить за счет вызова имеющегося в объекте глифа метода `level()` распознавание текущего вложенного уровня, а затем нарисовать соответствующую фигуру. Если бы мне пришлось создавать все это заново, то я обеспечил бы API выборкой правильного глифа для использования на каждом уровне вложенности.

Другая неприятность заключена в модели прямоугольника. Глифам позволено распределять вокруг себя дополнительное пространство с целью рисования дополнительных элементов в виде стрелок, подсветок или надписей. Это делается с помощью подмены методов, названных `pad_left()`, `pad_right()`, и т. д.

Все работает хорошо до тех пор, пока вы не определите новый класс глифа, который является наследником какого-нибудь старого класса, и у вас не возникнет необходимости корректировки дополнительного пространства с помощью дополнительных декоративных элементов. Производный класс должен позаботиться о том, чтобы узнать, сколько дополнительного пространства запрошено его родительским классом (за счет вызова унаследованного `pad`-метода), а затем добавить к этому значению свои собственные потребности. Это может оказаться не так-то просто сделать. Если бы мне пришлось это переделывать, то я просто отслеживал бы в его процедуре `draw_component()`, где глиф что-то рисует, и увеличивал, насколько нужно, ограничивающий его прямоугольник.

К сожалению, реализация обеих этих поправок приведет к существенным изменениям API глифа и потребует от кого-то, скорее всего от меня самого, переписать все существующие классы глифов, счет которым уже перевалил за шесть десятков, чтобы они не пришли в негодность. Поэтому в настоящее время я буду придерживаться того мнения, что модуль достаточно хорош, но никогда не сможет достичь совершенства. И это последний и, может быть, самый лучший из всех извлеченных мною уроков.

13 Конструкция генного сортировщика

Джим Кент (Jim Kent)

В этой главе рассматривается созданная мной средняя по размеру программа под названием Генный сортировщик – Gene Sorter. Объем кода этой программы больше, чем у большинства проектов, описанных в других главах, и составляет около 20 000 строк. Тем не менее в Gene Sorter есть ряд небольших, довольно привлекательных фрагментов, которые, на мой взгляд, в целом красивы легкостью восприятия, понимания и расширения. В этой главе я предлагаю вам краткий обзор возможностей программы Gene Sorter, выделяя некоторые наиболее важные части кода, а затем обсуждаю вопрос о том, как сделать программы, длина которых составляет более тысячи строк, более приятными и даже красивыми в работе.

Программа Gene Sorter помогает ученым быстро подвергать анализу примерно 25 000 генов в человеческом геноме и выискивать те из них, которые наиболее соответствуют предмету их исследования. Программа является частью веб-сайта <http://genome.ucsc.edu>, который также содержит множество других инструментов для работы с данными, созданными в рамках проекта по изучению генома человека – Human Genome Project. Конструкция программы Gene Sorter отличается простотой и гибкостью. В ней учтено множество уроков, извлеченных из разработки двух предыдущих поколений программ, которые вели в сети Интернет обработку биомедицинских данных. Программа с помощью CGI осуществляет сбор данных, поступающих от пользователей, создает запросы к базе данных MySQL и представляет результаты в формате HTML. Около половины программного кода размещено в библиотеках, которые используются совместно с другим инструментарием, представленным на веб-сайте genome.ucsc.edu.

Человеческий геном представлен цифровым кодом, который тем или иным образом содержит всю информацию, необходимую для построения человеческого организма, включая наиболее замечательный из всех органов – человеческий мозг. Информация хранится в трех миллиардах цепочек ДНК. Каждая цепочка может относиться к A, C, G или T виду.

Таким образом, на каждую цепочку приходится по два бита информации, или 750 Мб информации на весь геном.

Впечатляет то, что информация, необходимая для построения человеческого организма, может легко уместиться на флеш-карте в вашем кармане. Еще более впечатляющим является тот факт, что в результате эволюционного анализа множества геномов установлено, что в действительности востребовано только около 10 % всей этой информации. Остальные 90 % генома состоят прежде всего из остатков эволюционных экспериментов, которые зашли в тупик, и из помех, оставленных вирусоподобными элементами, известными как транспозоны.

В генах уже найдена большая часть задействованных на данный момент геномов.

Гены состоят из регулирующих элементов, которые определяют объем создаваемого генного продукта, и самого кода для создания генного продукта. Регуляция генов зачастую очень сложный процесс. Для разных типов клеток действуют различные гены. Одни и те же типы клеток используют в различных ситуациях разные гены.

Генные продукты также разнообразны. Весьма обширным и важным классом генной продукции является информационная РНК, которая затем трансформируется в белок. Эти белки включают молекулы рецепторов, которые позволяют клетке воспринимать состояние окружающей среды и взаимодействовать с другими клетками, ферменты, которые помогают преобразовывать питательные вещества в более пригодные для использования формы энергии, и транскрипционные факторы, которые управляют деятельностью других генов. При всей трудности этой работы, ученые идентифицировали около 90 % генов в геноме человека, всего свыше 20 000 генов.

Интерес большинства научно-исследовательских проектов связан всего лишь с несколькими десятками этих генов. Люди исследуют редкие генетические заболевания, изучая систему наследования болезни, чтобы связать ее с возможной областью отдельной хромосомы, состоящей из 10 000 000 цепочек. В последние годы ученые попытались связать области с 100 000 цепочками с наиболее распространенными болезнями, такими как диабет, которые по своей природе носят частично генетический характер.

Пользовательский интерфейс программы Gene Sorter

Программа Gene Sorter может собрать все известные гены в области ДНК, имеющей отношение к какому-нибудь заболеванию, в список возможных генов. Этот список показан в таблице на рис. 13.1 и включает суммарную информацию о каждом гене и ссылку на дополнительную информацию. Список возможных генов может быть отфильтрован, чтобы исключить гены, которые явно не относятся к делу, например гены, имеющие отношение только к почкам, в то время как исследователь занимается болезнями легких. Программа Gene Sorter может быть полезной и в других ситуациях, когда кто-то хочет просмотреть информацию, касающуюся более чем одного гена, к примеру, при изучении генов,

имеющих сходную структуру, или генов, о которых известно, что они имеют сходные функции. В настоящее время программа Gene Sorter доступна для работы с геномами человека, мыши, крысы, фруктовой мушки и свободноживущей нематоды (круглого червя).

Элементы управления в верхней части экрана определяют, какую версию какого генома нужно использовать. Расположенная ниже таблица содержит по одной строке на ген.

The screenshot shows the UCSC Human Gene Sorter interface. At the top, there are navigation links for 'Home', 'UCSC Human Gene Sorter', and 'Help'. Below that is a search bar with fields for 'genome' (set to 'Human'), 'assembly' (set to 'Mar, 2006'), 'search' (containing 'NM_003179'), and 'sort by' (set to 'Gene Distance'). There are also buttons for 'Configure', 'Filter now (0)', 'display' (set to '50'), 'output' (set to 'sequence'), and 'text'.

The main content is a table with the following columns: Name, VisiGene ID, Genomic tracks, Expression, Function, Disease, Chromosome, Start, End, and Description. The table lists 16 genes, starting with SYP at chrX:48,937,407 and ending with OTUD5 at chrX:48,682,134. The 'Description' column provides brief details about each gene's function.

Name	VisiGene ID	Genomic tracks	Expression	Function	Disease	Chromosome	Start	End	Description
SYP	181524					chrX	48,937,407		synaptophysin
LMO6	221					chrX	48,934,285		LIM domain only 6
PLP2	181508					chrX	48,916,798		protolipid protein 2 (colonic)
CACNA1P	174590					chrX	48,967,622		calcium channel, voltage-dependent, alpha 1F
FLJ21687	179636					chrX	48,909,468		PDZ domain containing, X chromosome
CCDC22	185850					chrX	48,586,613		coiled-coil domain containing 22
FOXP3	1768					chrX	49,001,293		forkhead box P3
GPKOW	n/a					chrX	48,862,156		G patch domain and KDW motifs
PPP1RJF	179945	n/a	n/a			chrX	49,022,141		protein phosphatase 1, regulatory (inhibitor)
WDR45	35149					chrX	48,832,019		WD repeat domain 45 (isoform 1)
PRAF2	175866					chrX	48,817,185		TM4 protein
JM11	185851					chrX	48,808,956		hypothetical protein LOC90060
TFE3	986					chrX	48,779,619		Hypothetical protein DKFZp76J11810.
GRIPAP1	30802					chrX	48,729,348		GRIP1 associated protein 1 isoform 1
KCND1	178025					chrX	48,708,389		potassium voltage-gated channel, Shal-related
OTUD5	62197					chrX	48,682,134		hypothetical protein LOC55593

Рис. 13.1. Главная страница программы Gene Sorter

Отдельные конфигурационные страницы управляют отображением в таблице тех или иных столбцов и их внешним видом. Страница фильтра может быть использована для фильтрации генов на основе любой комбинации значений столбцов.

Таблица может быть отсортирована несколькими способами. В данном случае она отсортирована по близости к избранному гену – SYP. Этот ген связан с освобождением медиаторов.

Поддержание диалога с пользователем по Интернету

Программа Gene Sorter – это CGI-скрипт. Когда пользователь нацеливает браузер на URL Gene Sorter (<http://genome.ucsc.edu/cgi-bin/hgNear>), веб-сервер запускает скрипт и посыпает по сети выходную информацию. Скрипт выводит HTML-форму. Когда пользователь щелкает на кнопке формы, веб-браузер посылает веб-серверу URL, который включает значения раскрывающихся меню

и других элементов управления, закодированных в виде серии пар переменная=значение. Веб-сервер еще раз запускает скрипт, передавая ему в качестве входящей информации пары переменная=значение. Затем в качестве ответа скрипт генерирует новую HTML-форму.

CGI-скрипт может быть написан на любом языке. Скрипт Gene Sorter представляет собой среднюю по объему программу, написанную на языке Си.

По сравнению с другими программами, взаимодействующими с компьютерными пользователями, CGI-скрипты имеют свои сильные и слабые стороны. CGI-скрипты обладают хорошей переносимостью и не требуют различных версий для поддержки пользователей на компьютерах Windows, Macintosh и Linux. С другой стороны, их интерактивность имеет весьма скромные показатели. Если не обращаться к JavaScript (который сам по себе вызывает довольно серьезные проблемы переносимости), дисплей будет обновляться, только когда пользователь щелкнет на кнопке и подождет секунду-другую появления новой веб-страницы. Тем не менее для большинства задач работы с геномом CGI предоставляет вполне приемлемую интерактивность и абсолютно стандартный пользовательский интерфейс.

Жизненный цикл CGI-скрипта очень ограничен. Он начинает работу в ответ на щелчок пользователя на каком-нибудь элементе управления и заканчивает ее генерацией веб-страницы. Вследствие этого скрипт не может сохранить долгосрочную информацию в программных переменных. Для самых простых CGI-скриптов вся необходимая информация хранится в парах переменная=значение (известных также как CGI-переменные).

Тем не менее для более сложных скриптов, таких как Gene Sorter, этого недостаточно, поскольку скрипт может нуждаться в запоминании результатов установки параметров, произведенной пользователем несколькими экранами ранее, но веб-сервер посыпает только CGI-переменные, соответствующие состоянию управляющих элементов самой последней из переданных страниц. Поэтому нашим CGI-скриптом нужен долговременный способ хранения данных.

Для хранения данных, невидимых в элементах управления формы, CGI предоставляет два механизма: скрытые CGI-переменные и cookie. В скрытых CGI-переменных данные хранятся в HTML, в форме <INPUT>-тегов, имеющих тип скрытых (hidden). При использовании cookie данные хранятся веб-браузером и посыпаются в http-заголовок.

Технология cookie при своем первом появлении вызывала споры, и некоторые пользователи предпочли их отключить. Тем не менее cookie могут сохраняться годами, а скрытые переменные исчезают сразу же, как только закрывается веб-страница. Ни cookie, ни скрытые переменные не могут хранить достаточно больших объемов данных. Точное количество зависит от веб-браузера, но обычно попытка сохранить с помощью этих механизмов более 4 Кб данных небезопасна.

Для использования cookie и скрытых переменных команда Gene Sorter разработала объект корзины — cart, который объединяет механизмы cookie и скрытых переменных с базой данных SQL. Cart поддерживает две таблицы, одна из которых связана с пользователем, а другая — с веб-сессиями. Таблицы имеют одинаковый формат, состоящий из ключевого поля, блоб-поля, содержа-

щего все пары переменная=значение в том самом формате, в котором они были переданы в URL, и полей, по которым отслеживаются временные параметры использования и счетчики обращений.

Ключ к таблице пользователя хранится в постоянно существующем cookie, а ключ к таблице сессий хранится в скрытой CGI-переменной. При запуске скрипта ищет в cookie ключ пользователя. Если он будет найден, то скрипт загружает связанную с ним пару переменная=значение в хэш-таблицу. Если cookie найден не будет, скрипт сгенерирует новый ключ пользователя, и хэш таблица останется пустой.

Затем скрипт ищет ключ сессии и загружает из него переменные, заменяя ими любые переменные, которые уже находятся в хэше. И наконец, любая новая CGI-переменная грузится на вершину того, что находится в хэше.

Ряд библиотечных процедур дает возможность скрипту считывать и записывать переменные в хэше корзины. Перед завершением своей работы скрипт обновляет таблицы базы данных текущим содержимым корзины. Если cookie на стороне пользователя запрещены, он все же будет иметь возможность взаимодействовать с программой Gene Sorter в течение отдельной сессии, поскольку ключ сессии не хранится в cookie. Обособление сессии от ключей пользователя также позволяет пользователю запускать Gene Sorter в двух отдельных окнах, которые будут работать, не мешая друг другу. Пользовательский уровень корзины позволяет программе Gene Sorter возобновлять работу с места последнего использования, даже после того как пользователь в промежутке переходил на другой веб-сайт.

В реализации Gene Sorter, представленной на genome.ucsc.edu, все CGI-скрипты на сайте пользуются одной общей корзиной. Соответственно, в корзине содержатся переменные даже более глобального типа, чем обычные переменные программы. Польза от этого проявляется довольно часто. Если пользователь в одной из наших программ заостряет свое внимание на геноме мыши, а не человека, то он, скорее всего, захочет воспользоваться информацией о мыши также и в других наших программах.

Однако поскольку наши программы разрастались, чтобы избежать случайного конфликта имен переменных корзины, мы приняли соглашение, что имена переменных корзины (если они не предназначались для глобального использования) начинались с имени использующего их CGI-скрипта. Таким образом, большинство имеющихся в Gene Sorter переменных корзины начинаются с префикса hgNear_. (Мы воспользовались вместо точки символом подчеркивания, поскольку точка будет конфликтовать с JavaScript.)

В целом корзина делает для Gene Sorter относительно простой поддержку для пользователей иллюзии непрерывности работы, даже если с каждым кликом запускаются отдельные экземпляры программы.

Короткий жизненный цикл CGI-скрипта имеет свои преимущества. В частности, CGI-скрипт не должен сильно заботиться об утечке памяти и закрытии файлов, поскольку при выходе из программы все очищается операционной системой. Это свойство особенно полезно для языка Си, который не имеет автоматического управления ресурсами.

Небольшой полиморфизм может иметь большое значение

Большинство программ, обладающих некоторой гибкостью, вероятнее всего будут содержать какой-либо полиморфный объект. Таблица, занимающая основное пространство главной страницы Gene Sorter, состоит из последовательности полиморфных столбцов-объектов.

Создание в Си полиморфных объектов не дается так же легко, как в большинстве современных объектно-ориентированных языков. Но это может быть сделано относительно простым способом, в котором вместо объекта используется struct, а вместо полиморфных методов — ссылки на функции. В примере 13.1 показана несколько сокращенная версия Си-кода для объекта столбца.

Пример 13.1. Структура столбца, пример полиморфного объекта, созданного на Си

```
struct column
/* Столбец большой таблицы. Главная структура данных для * hgNear. */
{
    /* Гарантия присутствия набора данных в каждом столбце. */
    struct column *next; /* Следующий столбец в списке. */
    char *name;           /* Имя столбца, не видимое пользователю. */
    char *shortLabel;     /* Обозначение столбца. */
    char *longLabel;      /* Описание столбца. */

    /* -- Методы -- */
    void (*cellPrint)(struct column *col, struct genePos *gp,
                      struct sqlConnection *conn);
    /* Вывод одной ячейки этого столбца в HTML. */

    void (*labelPrint)(struct column *col);
    /* Вывод обозначения в строке обозначений. */

    void (*filterControls)(struct column *col,
                           struct sqlConnection *conn);
    /* Вывод элементов управления для улучшенного фильтра. */

    struct genePos *(*advFilter)(struct column *col,
                                 struct sqlConnection *conn,
                                 /* Возврат перечня позиций для улучшенного фильтра. */

    /* Справочные таблицы используют следующие поля. */
    char *table;           /* Имя ассоциированной таблицы. */
    char *keyField;         /* Поле идентификатора гена ассоциированной
                           таблицы.*/
    char *valField;         /* Поле значения ассоциированной таблицы.*/

    /* Наряду с полями подстановки ассоциированные таблицы используют
     следующие запросы. */
    char *queryFull;        /* Запрос, возвращающий 2 столбца ключ/значение. */
    char *queryOne;         /* Запрос, возвращающий значение по заданному
                           ключу. */
    char *invQueryOne;      /* Запрос, возвращающий ключ по заданному
                           значению. */
}:
```

Структура начинается с данных, совместно используемых всеми типами столбцов. Затем следуют полиморфные методы. И в завершение раздел содержит данные, определяемые спецификой типа.

Каждый объект столбца содержит пространство для данных всех типов столбцов. Можно было бы избежать пустой траты пространства, используя объединение или некоторые другие родственные механизмы. Но это усложнило бы использование полей специфического типа, а поскольку всего используется менее 100 столбцов, общее сэкономленное пространство не превысило бы нескольких килобайт.

Основная функциональная часть программы находится в методах столбцов. Столбцы «знают», как получить данные о конкретном гене в виде строки или в виде HTML-кода. Столбцы способны искать гены, поскольку данные столбца могут содержать простую строку поиска. Столбцы также осуществляют интерактивное управление фильтрацией данных и содержат саму процедуру фильтрации.

Столбцы создаются специализированной процедурой на основе информации, содержащейся в файлах columnDb.ra. Выборка из одного из таких файлов показана в примере 13.2. Все записи columnDb содержат поля, которые дают описание имени столбца, в виде видимых пользователю коротких и длинных обозначений, заданное по умолчанию местоположение столбца в таблице (его приоритет), сведения об установке по умолчанию видимости столбца для пользователя и о типе поля. Тип поля управляет составом методов, имеющихся у столбца. В файлах могут быть и дополнительные столбцы, некоторые из которых могут быть специфического типа. Во многих случаях SQL используется для осуществления запросов к таблицам базы данных, связанных со столбцом, включенных в запись columnDb наряду с URL гиперссылки на каждый элемент столбца.

Пример 13.2. Раздел файла columnDb.ra, в котором содержатся метаданные столбцов

```
name proteinName
shortLabel UniProt
longLabel UniProt (SwissProt/TrEMBL) Protein Display ID
priority 2.1
visibility off
type association kgXref
queryFull select kgID.spDisplayID from kgXref
queryOne select spDisplayId.spID from kgXref where kgID = '%s'
invQueryOne select kgID from kgXref where spDisplayId = '%s'
search fuzzy
itemUrl http://us.expasy.org/cgi-bin/niceprot.pl?%s

name proteinAcc
shortLabel UniProt Acc
longLabel UniProt (SwissProt/TrEMBL) Protein Accession
priority 2.15
visibility off
type lookup kgXref kgID spID
search exact
itemUrl http://us.expasy.org/cgi-bin/niceprot.pl?%s
```

```

name refSeq
shortLabel RefSeq
longLabel NCBI RefSeq Gene Accession
priority 2.2
visibility off
type lookup knownToRefSeq name value
search exact
itemUrl http://www.ncbi.nlm.nih.gov/entrez/query.
fcgi?cmd=Search&db=Nucleotide&term=%s&doptcmdl=GenBank&tool=genome.ucsc.edu

```

Файл `columnDb.ra` имеет довольно простой формат: по одному полю на строку, а записи разделены пустыми строками. Каждая строка начинается с имени поля, а в оставшейся части строки содержится его значение.

Такой же простой линейный формат используется для большого количества метаданных веб-сайта `genome.ucsc.edu`. Было время, когда мы рассматривали использование индексированных версий этих файлов в качестве альтернативы реляционным базам данных (`.ra` заменяло слова `relational alternative`). Но реляционные базы данных располагают огромным количеством полезных инструментальных средств, поэтому мы решили хранить основную массу своих данных в реляционном виде. Тем не менее файлы с расширением `.ra` очень легко читаются, редактируются и подвергаются синтаксическому анализу, поэтому они продолжают использоваться в таких приложениях, как это.

Файлы `columnDb.ra` размещены в каталогах с трехуровневой иерархией. В корневом каталоге находится информация о столбцах, появляющихся для всех организмов. Средний уровень содержит информацию, специфическую для каждого организма. В соответствии с нашим пониманием прогресса расшифровки генома отдельного организма у нас будут различные сборки его ДНК-последовательностей. Нижний уровень содержит информацию, характерную для сборки.

Код,читывающий `columnDb`, создает хэш, состоящий из вложенных хэшей, в котором внешний хэш в качестве ключей использует имена столбцов, а у внутренних хэшей ключами служат имена полей. Информация на низших уровнях может содержать абсолютно новые записи, а также дополнять или подменять отдельные поля записей, первоначально определенных на более высоком уровне.

Некоторые типы столбцов напрямую соответствуют столбцам в реляционной базе данных. Столбцы типа `lookup` ссылаются на таблицу, которая содержит проиндексированное поле идентификатора гена (`gene ID`), для каждого идентификатора не может быть более одной строки. Стока `type` содержат таблицу, поле `gene ID` и поле, отображаемое в столбце. Столбцы `proteinAcc` и `refSeq` в примере 13.2 относятся к типу `lookup`.

Если реляционная таблица может содержать более одной строки на ген, ей присваивается тип `association`. Связи одного гена с несколькими значениями отображаются в Gene Sorter в виде списка с запятыми в качестве разделителей. Связи включают SQL-код выборки данных либо для одного гена за один запрос (`queryOne`), для всех генов (`queryFull`), либо для генов, связанных с определенным значением (`invQueryOne`). SQL-запрос `queryOne` возвращает два значения, одно для отображения в Gene Sorter, а другое для использования в качестве гиперссылки, хотя это может быть одно и то же значение.

Большинство столбцов в Gene Sorter имеют типы `lookup` или `association`, и если взять любую реляционную таблицу, проиндексированную по ключу идентификатора гена (`gene ID`), то сделать из нее столбцы Gene Sorter будет совсем нетрудно.

Другие столбцы, такие как столбцы экспрессии генов, являются относительно сложными объектами. На рис. 13.1 показан столбец экспрессии генов в виде раскрашенных прямоугольников, расположенных под названиями различных органов, таких как мозг, печень, почка и т. д. Цвет отображает количество информационных РНК гена, найденных в определенных органах, по сравнению с уровнем информационных РНК во всем организме. Красный цвет показывает уровень экспрессии выше среднего, а черный — средний уровень экспрессии.

Весь набор информации об экспрессии гена от мозга зародыша (*fetal brain*) до яичек (*testis*) на рис. 13.1 рассматривается в одном столбце Gene Sorter. Для лучшей читаемости с точки зрения HTML этот столбец разбит на три столбца, с серыми полосами между группами из пяти органов.

Фильтрация, оставляющая только значимые гены

Фильтры — одна из самых сильных особенностей сортировщика генов. Они могут применяться к каждому столбцу в целях просмотра генов определенной предназначенности. Например, фильтр столбца экспрессии генов может быть использован для поиска генов, проявляющих экспрессию в мозге, но не в других тканях. Фильтр в столбце позиции генома (*genome position*) может найти гены X-хромосомы. Комбинации этих фильтров может найти гены, относящиеся к мозгу, находящиеся в X-хромосоме. Особый интерес к этим генам могут проявить исследователи аутизма, поскольку условия его возникновения в довольно большой степени связаны с половыми признаками.

У каждого столбца имеется два метода фильтрации: `filterControls` — для создания HTML-кода, отображающего фильтр в пользовательском интерфейсе, и `advFilter` — для запуска фильтра. Эти два метода связываются друг с другом через корзину переменных, в которой используется соглашение об именах, заключающееся в использовании в качестве префикса имени определенной переменной имени программы, букв `as` и имени столбца. Таким образом, разные столбцы одного и того же типа имеют разные переменные корзины, а переменные фильтра могут иметь характерные отличия от других переменных. В системе фильтрации широко используется вспомогательная процедура по имени `cartFindPrefix`, возвращающая список всех переменных с заданным префиксом.

Фильтры выстраиваются в виде цепочки. Вначале программа создает список всех генов. Затем она проверяет корзину на установку тех или иных фильтров. Если фильтры установлены, то они вызываются для каждого столбца. Первый фильтр получает на входе полный список генов. Последующие фильтры начинают работу с данными, полученными на выходе предыдущего фильтра. Порядок применения фильтров не имеет значения.

Фильтры являются основным кодом Gene Sorter, влияющим на его быстродействие. Большая часть кода воздействует на информацию всего лишь о 50 или 100 генах, а фильтры обрабатывают десятки тысяч таких информационных наборов. Чтобы поддерживать приемлемое время ответной реакции, фильтры должны затрачивать менее 0,0001 секунды на один ген. Современные процессоры работают настолько быстро, что для них показатель в 0,0001 секунду не является каким-то существенным ограничением. Тем не менее скорость поиска на жестком диске все еще занимает около 0,005 секунд, поэтому фильтр должен избегать вынужденного поиска.

Большинство фильтров начинают свою работу с проверки корзины на наличие установленных для них переменных, и, если они отсутствуют, фильтры сразу же возвращают входной список в неизменном виде. Затем фильтрычитывают таблицы, связанные со столбцами. Считывание всей таблицы позволяет избежать возможного вынужденного поиска на диске для каждого элемента, и хотя для обработки нескольких генов это замедляет процесс, зато он существенно ускоряется при обработке большого набора генов.

Гены, прошедшие фильтр, помещаются в хэш, ключом для которого служит идентификатор гена (gene ID). В заключение фильтр вызывает процедуру под названием `weedUnlessInHash`, которая осуществляет циклический перебор генов во входных данных, чтобы определить их присутствие в хэше, и, если они присутствуют, копирует их в выходные данные. В результате получается быстрая и гибкая система с относительно небольшим количеством кода.

Теория красоты кода в крупных формах

Gene Sorter одна из наиболее красивых программ по конструкторскому замыслу и программному коду из всех, над которыми мне приходилось работать. Большинство наиболее важных частей системы, включая корзину, каталог .г-файлов и интерфейс к базе данных геномов, представлены уже в своем втором или третьем варианте, учитывающем уроки предшествующих программ. Структура объектов программы красиво вписывается в основные компоненты пользовательского интерфейса и реляционных баз данных. Используемые алгоритмы просты, но эффективны, и обладают неплохим соотношением между скоростью работы, использованием памяти и сложностью программного кода. По сравнению сопоставимыми по размеру программами, в этой программе было обнаружено очень мало ошибок. Работа основного кода может быть ускорена усилиями других людей, которые относительно быстро могут внести в него свой вклад.

Программирование является родом человеческой деятельности, и, наверное, наиболее ограничивающим ресурсом при программировании является наша человеческая память. Обычно в своей краткосрочной памяти мы можем хранить с полдесятка фактов. Для большего количества требуется подключить нашу долговременную память. А система долговременной памяти имеет невероятно огромный объем, но ввод в нее сведений происходит относительно медленно,

и мы не можем осуществлять из нее произвольную выборку, поскольку она работает только ассоциативным образом.

В то время как структура программы не более чем в несколько сотен строк может быть продиктована алгоритмическими или машинными соображениями, структура более объемных программ должна быть продиктована человеческим фактором, по крайней мере, если мы предполагаем чью-то продуктивную работу по их обслуживанию и расширению в долговременной перспективе.

В идеале нужно лишь понять, что фрагмент программного кода должен поместиться на одном экране. Если он не поместится, то читатель будет вынужден в лучшем случае перескакивать от экрана к экрану, в надежде понять смысл этого кода. Если код непростой, то читатель, скорее всего, забудет о том, что было определено на каждом экране за то время, пока будет возвращаться к исходному экрану, и ему придется запоминать большой объем кода, пока он не разберется во всех его частях. Разумеется, это замедлит работу программистов, к тому же многие из них почувствуют глубокое разочарование.

Чтобы сделать код в отдельных местах более понятным, нужно уделить большое внимание удачному подбору имен. Вполне допустимо создание нескольких локальных переменных (не больше, чем может поместиться в краткосрочной памяти) с именами в одну-две буквы. Все остальные имена должны быть словами, короткими фразами или общепринятыми (и короткими) аббревиатурами. В большинстве случаев должна быть возможность различать назначение переменной или функции только по одному ее имени.

В наши дни с имеющейся в распоряжении модной интегрированной средой разработки читатель может щелчком мыши перейти из того места, где используется обозначение, туда, где оно определено. Тем не менее нам хотелось написать код так, чтобы у пользователей возникала потребность перехода к месту определения обозначения только в том случае, если он проявит любопытство к подробностям. Нам не хотелось заставлять его следовать по нескольким ссылкам, чтобы понять назначение каждой строки.

Имена должны быть не слишком длинными, но и не слишком короткими, хотя многие программисты под влиянием математического описания алгоритмов и таких вредных вещей, как венгерская система записи, ошибочно склоняются к короткому варианту. Чтобы придумать удачное имя, нужно время, но это время будет потрачено не зря.

Для локальной переменной удачно подобранное имя может вполне заменить документирование. На рис. 13.3 показана весьма продуманно выполненная функция, принадлежащая программе Gene Sorter. Она фильтрует связи в соответствии с условием, в котором могут содержаться символы-заменители. (Есть также более простой и быстрый метод, обрабатывающий только строгие соответствия.) Код помещается на одном экране, что для функций является безусловным, но не всегда возможным преимуществом.

Пример 13.3. Метод фильтрации для столбцов типа association, обрабатывающий символы-заменители

```
static struct genePos *wildAssociationFilter(
    struct s1Name *wildList, boolean orLogic, struct column *col,
```

```

struct sqlConnection *conn, struct genePos *list)
/* Фильтрация связей, соответствующих любому списку символов-заменителей. */
{
/* Групповые связи по генным идентификаторам. */
struct assocGroup *ag = assocGroupNew(16);
struct sqlResult *sr = sqlGetResult(conn, col->queryFull);
char **row;
while ((row = sqlNextRow(sr)) != NULL)
    assocGroupAdd(ag, row[0], row[1]);
sqlFreeResult(&sr);

/* Поиск соответствующих связей и помещение их в passHash. */
struct hash *passHash = newHash(16); /* Hash of items passing filter */
struct genePos *gp;
for (gp = list; gp != NULL; gp = gp->next)
{
    char *key = (col->protKey ? gp->protein : gp->name);
    struct assocList *al = hashFindVal(ag->listHash, key);
    if (al != NULL)
    {
        if (wildMatchRefs(wildList, al->list, orLogic))
            hashAdd(passHash, gp->name, gp);
    }
}

/* Построение отфильтрованного списка, завершение работы и возвращение. */
list = weedUnlessInHash(list, passHash);
hashFree(&passHash);
assocGroupfree(&ag);
return list;
}

```

Прототип функции сопровождается комментарием в одно предложение, в котором дается краткое описание того, что в ней делается. Код внутри функции разбит на «параграфы», каждый из которых начинается с комментария с кратким описанием того, что делается в блоке.

Программисты могут читать код этой функции с несколько иным уровнем детализации. Одним обо всем, что им нужно, расскажут имена. Другим понадобится прочитать также и вводные комментарии. А кто-то, игнорируя код, пропустит все комментарии. Те же, кто интересуется всеми подробностями, будут вычитывать каждую строку.

Поскольку человеческая память имеет ярко выраженный ассоциативный характер, если функция прочитана на одном уровне детализации, то прочитать ее на более высоком уровне обычно достаточно, чтобы вспомнить более детализированные уровни. Но это явление носит частичный характер, поскольку более высокий уровень формирует структуру организации ваших воспоминаний о функции в тот самый момент, когда вы занимаетесь чтением на более низком уровне.

В общем, чем объемнее объект программирования, тем большего документирования он заслуживает. Переменная требует хотя бы слова, функция — как минимум предложения, а более крупные элементы, такие как модули или

объекты, возможно, и целого абзаца. Очень полезно сопровождать всю программу несколькими страницами документации, которая дает ее краткий обзор.

Но в объеме документации нужно соблюдать разумную меру. Документация бесполезна, если ее никто не читает, а люди не любят читать длинные тексты, особенно скучные по содержанию.

Люди лучше запоминают какие-то важные моменты, хотя некоторые обладают даром (или проклятием) хорошей памяти и на мелочи. Важную роль играют те слова, которые используются при программировании в качестве имен, но выбор стиля `varName`, `VarName`, `varname`, `var_name`, `VARNAME`, `vrblnam` или `Variable_Name` особой роли не играет. Важно придерживаться единого соглашения, чтобы программист не тратил зря времени и не напрягал память, чтобы вспомнить, какой стиль использовать в той или иной ситуации.

Другими ключами к поддержанию кода в понятном для всех виде являются следующие положения.

- Сужайте область видимости насколько это возможно. Никогда не используйте глобальные переменные, когда можно воспользоваться объектными переменными, и никогда не пользуйтесь объектными переменными, когда можно обойтись локальными.
- Минимизируйте побочные эффекты. В частности, избегайте *изменения* любых переменных, если только они не являются возвращаемым значением функции. Функция, которая придерживается этого правила, называется «повторно используемой» и считается элементом красоты программирования. Ее не только просто понять, она автоматически становится поточно-ориентированной и может быть использована в рекурсивном режиме. Кроме того что код с минимальным количеством побочных эффектов хорошо читается, его также проще повторно использовать в различных контекстах.

Сегодня многие программисты хорошо осведомлены о негативном влиянии глобальных переменных на возможность повторного использования кода. Другим фактором, препятствующим повторному использованию кода, является зависимость от структуры данных. Объектно-ориентированный стиль программирования в этом отношении иногда может вести к неприятным последствиям. Если подходящий код встроен в объект в виде метода, то чтобы им воспользоваться, нужно создавать этот объект. Для некоторых объектов эта задача может быть существенно затруднена.

Функция, не встроенная в объект и принимающая в качестве параметров данные стандартных типов, имеет значительно более высокие шансы использования в разнообразной обстановке, чем метод, глубоко встроенный в сложную иерархию объекта. К примеру, хотя ранее упомянутая функция `weedUnlessInHash` написана для использования объектами столбцов программы Gene Sorter, она была сконструирована независимой от какого-нибудь столбца. Поэтому теперь эта небольшая и полезная функция может также найти себе применение и в другой обстановке.

Вывод

Эта глава была посвящена самым значительным из написанных мной фрагментов программного кода. Программа несет полезную службу в области биомедицинских исследований. Система с использованием корзины в какой-то мере упростила создание интерактивной программы, работающей через Интернет, даже при том, что в ней используется CGI-интерфейс.

Обе модели программы, как пользовательская, так и программистская, построены на идее большой таблицы, имеющей по одной строке на каждый ген и переменное количество столбцов, с которых могут быть представлены различные типы данных.

Хотя программа Gene Sorter написана на Си, код столбцов выполнен в открытой, полиморфной, объектно-ориентированной конструкции. Новые столбцы могут добавляться путем редактирования простых текстовых файлов и не требуют дополнительного программирования, и эти самые файлы помогают упростить отдельной версии программы работу с различными базами данных геномов, связанных с множеством организмов.

Конструкция программы позволяет минимизировать поиск на жестком диске, который по-прежнему является узким местом, существенно отставая от скоростей работы процессора и памяти. Код написан с прицелом на хорошую читаемость и возможность повторного использования. Я надеюсь, что вы найдете некоторые принципы, на которых построена программа, полезными для создания ваших собственных программ.

14 Как первоклассный код развивается вместе с аппаратным обеспечением (на примере Гауссова исключения)

Джек Донгарра (*Jack Dongarra*) и Пётр Лушчек (*Piotr Luszczek*)

Растущие возможности компьютеров, имеющих передовую архитектуру и приемлемую цену, существенно влияют на все сферы научных расчетов. В этой главе на примере подробного исследования одного из простых, но весьма важных алгоритмов математического программного обеспечения – Гауссова исключения для решения систем линейных уравнений – мы покажем, что создателям компьютерных алгоритмов необходимо заняться своевременной и основательной их адаптацией к произошедшем в компьютерной архитектуре изменениям.

На уровне приложений научные положения должны быть отражены в математических моделях, которые, в свою очередь, находят свое выражение в алгоритмах, а те в конечном счете превращаются в программный код. На уровне программного обеспечения существует постоянное внутреннее противоречие между производительностью и переносимостью, с одной стороны, и разборчивостью основного кода, с другой. Мы проведем исследование этих проблем и рассмотрим возникшие со временем компромиссы. Линейная алгебра, в частности решение системы линейных уравнений, лежит в основе многих вычислений при производстве научных расчетов. Эта глава сфокусирована на некоторых последних разработках в программном обеспечении линейной алгебры, предназначенных для работы на компьютерах, обладающих передовой архитектурой, появившейся за последние десятилетия.

Существует два обширных класса алгоритмов: для плотных и для разреженных матриц. Матрица называется *разреженной*, если содержит существенное количество нулевых элементов. Для разреженных матриц путем применения специализированных алгоритмов и хранилищ можно достичь значительной экономии пространства и времени выполнения. Чтобы сузить предмет разговора и избежать усложнений, мы ограничимся рассмотрением задачи, связанной

с *плотными матрицами* (плотной считается матрица с небольшим количеством нулевых элементов).

Большая часть работ по созданию программного обеспечения для решения задач линейной алгебры на компьютерах с передовой архитектурой мотивирована потребностями сделать доступными решения объемных задач на самых быстрых компьютерах.

В этой главе мы рассмотрим разработку стандартов программного обеспечения, предназначенного для решения задач линейной алгебры, унифицированные модули программных библиотек и аспекты конструкции алгоритмов, на которые оказывают влияние возможности параллельной реализации. Мы объясним, чем мотивирована эта работа, и затронем вопросы дальнейшего направления ее развития.

В качестве типичного примера подпрограммы решения задач плотных матриц мы рассмотрим Гауссово исключение, или LU-факторизацию. Этот анализ, охватывающий аппаратный и программный прогресс за последние 30 лет, выветрит наиболее важные факторы, которые должны быть приняты во внимание при проектировании программ линейной алгебры для компьютеров с передовой архитектурой. Мы воспользуемся в качестве иллюстрации подпрограммами разложения на множители не только из-за их относительно просты, но и с учетом их важности для ряда научных и инженерных приложений, в которых используются методы предельного элемента. Это, в частности, приложения по расчетам электромагнитного рассеивания и приложения по решению задач гидрогазодинамики.

В последние 30 лет наблюдалась высокая активность в области разработки алгоритмов и программного обеспечения для решения задач линейной алгебры. Цель достижения высокой производительности переносимого на различные платформы кода во многом была достигнута за счет идентификации в подпрограммах основных компонентов линейной алгебры, Basic Linear Algebra Subprograms (BLAS). Мы рассмотрим представленные в последовательных уровнях BLAS библиотеки LINPACK, LAPACK и ScaLAPACK.

Информация о подробном описании этих библиотек находится в конце главы, в разделе «Дополнительная литература».

Влияние компьютерной архитектуры на матричные алгоритмы

При проектировании эффективных алгоритмов линейной алгебры для компьютеров с передовой архитектурой основной интерес касается хранения и извлечения данных. Проектировщики стремятся свести к минимуму частоту перемещения данных между различными уровнями иерархии памяти.

Если данные находятся в регистрах или в самом быстром кэше, то вся необходимая для этих данных обработка должна быть проведена до того, как данные будут отосланы назад, в основную память. Поэтому при использовании в наших реализациях как векторизации, так и параллелизма, основной алгоритмический подход связан с применением алгоритмов *блочного разбиения*, особенно

в сочетании с ядрами, имеющими тонкую настройку на выполнение матрично-векторных и матрично-матричных операций (Уровни Level-2 и Level-3 BLAS).

Блочное разбиение означает, что данные поделены на блоки, каждый из которых должен помещаться в кэш-памяти или в векторном регистровом файле.

В этой главе рассматриваются следующие компьютерные архитектуры:

- векторные машины;
- RISC-компьютеры, обладающие кэш-иерархиями;
- параллельные системы с распределенной памятью;
- многоядерные процессоры.

Векторные машины появились в конце 70-х – начале 80-х годов прошлого века и могли за один такт выполнить одну операцию над относительно большим количеством операндов, сохраненных в векторных регистрах. Представление матричных алгоритмов в виде векторно-векторных операций было их естественной подгонкой для этого типа машин. Однако некоторые векторные конструкции обладали ограниченной возможностью по загрузке и хранению векторных регистров в основной памяти. Технология, получившая название *сплайнение*, позволила обойти эти ограничения за счет перемещения данных между регистрами перед обращением к оперативной памяти. Сплайнение требует переделки линейной алгебры в термины матрично-векторных операций.

На рубеже 80–90-х годов прошлого столетия появились RISC-компьютеры. Хотя их тактовую частоту можно было сопоставить с тем же показателем векторных машин, в вычислительной скорости они уступали, поскольку не имели векторных регистров. Другой их недостаток заключался в организации глубокой иерархии памяти с несколькими кэш-уровнями, предназначенными для компенсации нехватки пропускной способности, которая, в свою очередь, была вызвана главным образом ограниченным количеством банков памяти. Последующий успех этой архитектуры обычно приписывается правильной ценовой политике и поразительному росту производительности в течение долгого времени, что соответствовало предсказаниям закона Мура. С появлением RISC-компьютеров алгоритмы линейной алгебры в очередной раз нуждались в переделке. На сей раз их разработка требовала максимального проявления матрично-матричных операций, что гарантировало улучшение многократного использования кэш-памяти.

Естественным способом достижения еще больших уровней производительности как на векторных, так и на RISC-процессорах было объединение их в сеть для совместной работы над решением задач, превышающих по объему те задачи, которые решались на одном процессоре. По этому пути развивались многие аппаратные конфигурации, поэтому матричные алгоритмы должны были снова спасать за этим развитием. Довольно быстро обнаружилось, что хорошая локальная производительность должна была сочетаться с хорошим глобальным разбиением матриц и векторов.

При любых обычных способах разбиения матричных данных тут же возникали проблемы масштабируемости, продиктованные так называемым законом Амдала, основанным на наблюдении, что время, затрачиваемое на последовательную часть вычисления, обуславливает тот минимум, который устанавливает

предел общего времени выполнения, ограничивая тем самым преимущества, достигаемые за счет параллельной обработки. Иными словами, если большинство вычислений не могут производиться независимо, то точка сокращения отдачи уже достигнута, и добавление процессоров к аппаратной связке не приведет к ускорению процесса.

Класс многоядерных архитектур в упрощенном представлении сводится к симметричной многопроцессорной обработке (symmetric multiprocessing, SMP) и многоядерным машинам, выполненным на одном кристалле. Возможно, это не совсем верное упрощение, поскольку SMP-машины обычно располагают лучшей системой памяти. Но в отношении матричных алгоритмов оба перечисленных типа выдают довольно хорошую производительность при использовании очень похожих алгоритмических подходов: сочетание многократного использования локальной кэш-памяти и независимого вычисления с явно выраженным управлением той зависимостью, которая существует в отношении данных.

Декомпозиционный подход

В основе решений плотных линейных систем лежит декомпозиционный подход. Основная идея заключается в следующем: если задача решается в отношении матрицы A , то нужно осуществить разложение на множители или провести декомпозицию A , чтобы получить результат в виде простых матриц, в отношении которых задача решается намного проще. Тем самым проблема вычисления делится на две части: сначала определяется соответствующая декомпозиция, а затем она используется для решения имеющейся задачи.

Рассмотрим задачу решения линейной системы:

$$Ax = b$$

где A — невырожденная матрица n -ного порядка. Декомпозиционный подход начинается с исследования возможности разложения A на множители вида:

$$A = LU$$

где L — нижняя треугольная матрица (матрица, содержащая выше диагонали одни нули), включающая элементы диагонали, а U — верхний треугольник (со всеми нулями ниже диагонали). В ходе процесса декомпозиции диагональные элементы A (называемые ведущими элементами) используются для разделения элементов ниже диагонали. Если в матрице A имеются нулевые ведущие элементы, процесс будет прерван из-за ошибки деления на нуль. Кроме того, маленькие значения ведущих элементов чрезмерно увеличивают числовые ошибки процесса. Поэтому для числовой стабильности метод должен провести обмен строк матрицы или удостовериться, что ведущие элементы имеют максимально возможное (по абсолютной величине) значение. Это наблюдение приводит к матрице перестановки строк P и изменяет вид разложения на множители на следующий:

$$P^T A = LU$$

Затем решение может быть записано в следующем виде:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{Pb}$$

который предлагает следующий алгоритм решения системы уравнений:

- 1) разложение \mathbf{A} на множители;
- 2) решение системы $\mathbf{Ly} = \mathbf{Pb}$;
- 3) решение системы $\mathbf{Ux} = \mathbf{y}$.

Подход к матричным вычислениям за счет декомпозиции оказался весьма полезным по нескольким причинам. Прежде всего, он разбивает вычисления на две стадии: вычисление декомпозиции и последующее использование декомпозиции для решения рассматриваемой задачи. К примеру, это может пригодиться при разных правых сторонах, к решению которых нужно подходить по-разному. Требуется только однократное разложение матрицы на множители и многократное его использование для различных правых сторон. Это очень важно, поскольку разложение на множители матрицы \mathbf{A} , шаг 1, требует $O(n^3)$ операций, а решения, заключенные в шагах 2 и 3, — только $O(n^2)$ операций. Другой аспект силы алгоритма заключается в хранилище: множители L и U не требуют какого-то дополнительного хранилища и могут использовать то самое пространство памяти, которое вначале было занято массивом \mathbf{A} .

Чтобы рассмотреть превращение этого алгоритма в программный код, мы представили только самую интенсивную вычислительную часть процесса, заключающуюся в шаге 1, в разложении матрицы на множители.

Простая версия

Для первой версии мы представляем простую и понятную реализацию LU-факторизации. Она состоит из $n-1$ шагов, каждый из которых, как показано на рис. 14.1, представляет все больше нулей ниже диагонали.

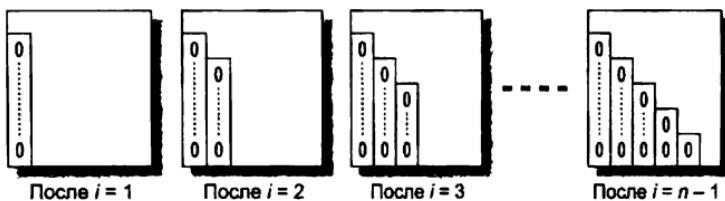


Рис. 14.1. LU-факторизация

В качестве инструмента, обучающего Гауссову исключению, часто используется система MATLAB. Она содержит язык сценариев (тоже называемый MATLAB), который значительно облегчает разработку матричных алгоритмов. Для людей, знакомых с другими языками сценариев, этот язык может показаться слишком необычным, поскольку он ориентирован на обработку многомерных

массивов. В примере кода мы воспользуемся следующими уникальными особенностями языка

- Оператором перестановки для векторов и матриц: ' (одинарная кавычка).
- Индексированием матриц, определяемым следующим образом:
 - элементарные целочисленные значения: $A(m, k)$;
 - диапазоны: $A(k:n, k)$;
 - другие матрицы: $A([k:m], :)$.
- Встроенными матричными функциями, среди которых `size` (возвращает размерность матрицы), `tril` (возвращает нижнюю треугольную часть матрицы), `triu` (возвращает верхнюю треугольную часть матрицы) и `eye` (возвращает единичную матрицу, содержащую только нулевые элементы, за исключением диагонали, содержащей одни единицы)

Наша простая реализация показана в примере 14.1.

Пример 14.1. Простой вариант (код на MATLAB)

```
function [L,U,p] = lutx(A)
%LUTX Треугольная факторизация, учебная версия
% [L,U,p] = lutx(A) создает модуль нижней треугольной матрицы L.
% верхней треугольной матрицы U. и вектор перестановки p.
% так что L*U = A(p,:)

[n,n] = size(A);
p = (1:n)';

for k = 1:n-1

    % Нахождение индекса 'm' наибольшего элемента 'r' ниже диагонали в k-том
    % столбце
    [r,m] = max(abs(A(k+1:n,k)));
    m = m+k-1; % корректировка 'm' под глобальный индекс

    % Пропуск исключения, если столбец нулевой
    if (A(m,k) == 0)

        % Замена ведущей строки
        if (m ~= k)
            A([k m],:) = A([m k],:); % обмен строк 'k' и 'm' матрицы 'A'
            p([k m]) = p([m k]); % обмен 'k' и 'm' в векторе 'p'
        end

        % Вычисление множителей
        i = k+1:n;
        A(i,k) = A(i,k)/A(k,k);

        % Обновление остатка матрицы
        j = k+1:n;
        A(i,j) = A(i,j) - A(i,k)*A(k,j);
    end
end
```

```

; Разделение результата
L = tril(A, -1) + eye(n,n);
U = triu(A);

```

Алгоритм, представленный в примере 14.1, ориентирован на обработку строк в том смысле, что мы берем скалярный множитель «ведущей» строки и прибавляем его к строкам, расположенным ниже, чтобы поместить нули ниже диагонали. Привлекательность алгоритма заключается в его схожести с математической записью. Следовательно, он является предпочтительным способом первоначального изучения алгоритма, который дает возможность студентам быстро превращать формулы в работоспособный код.

Но эта привлекательность имеет свою цену. В 70-х годах прошлого столетия Фортран был языком научных расчетов. Двумерные массивы хранились в Фортране в виде столбцов. Построчный доступ к массиву мог повлечь за собой последовательную ссылку на участки памяти, отделенные друг от друга значительным интервалом, зависящим от размера объявленного массива. В процессе выполнения программы в среде Фортрана, при достаточно большой матрице и алгоритме, ориентированном на обработку строк, могло быть сгенерировано слишком большое количество перестановок страниц. Клив Молер (Cleve Moler) указывал на это в те же 70-е годы (см. список дополнительной литературы в конце главы).

Во избежание подобной ситуации нужно было просто заменить порядок использования *i* и *j* в самых вложенных циклах. Такое незамысловатое изменение привело к 30-процентной экономии реального времени при решении задач с показателем размера, равным 200, на машинах IBM 360/67. Красота в данном случае была заменена эффективностью за счет использования менее очевидного порядка следования циклов и более запутанного (по сегодняшним меркам) языка.

Подпрограмма DGEFA библиотеки LINPACK

Проблемы производительности с версией кода на MATLAB продолжались, в то время как в середине 70-х годов XX века для научных вычислений стала доступна векторная архитектура. В этой архитектуре использовалась конвейерная обработка за счет запуска математических операций над массивами данных в одновременном или конвейерном режиме. Многие имеющиеся в линейной алгебре алгоритмы довольно легко могли быть векторизованы.

Поэтому в конце 70-х годов были предприняты усилия по стандартизации векторных операций для их использования в научных расчетах. Идея заключалась в определении некоторых простых, часто используемых операций и их реализации на различных системах в целях достижения переносимости и эффективности.

Этот пакет появился под названием Level-1 Basic Linear Algebra Subprograms (BLAS) или Level-1 BLAS.

Термин Level-1 означал векторно-векторные операции. Далее мы увидим, что Level-2 (матрично-векторные операции) и Level-3 (матрично-матричные операции) также играют весьма существенную роль.

В 70-е годы алгоритмы плотной линейной алгебры были методично реализованы в рамках проекта LINPACK, который представлял собой набор фортрановских подпрограмм, проводящих анализ и решение линейных уравнений и линейных задач метода наименьших квадратов. Этот пакет решает линейные системы с общими, ленточными, симметрично неопределенными, симметрично положительно определенными, треугольными и трехдиагональными квадратными матрицами. Кроме того, пакет вычисляет QR-разложение и разложение прямоугольных матриц по особым значениям и применяет их для задач наименьших квадратов.

LINPACK использует алгоритмы, ориентированные на работу со столбцами, которые повышают эффективность за счет сохранения местоположения ссылок. Под ориентацией на столбцы мы имеем в виду то обстоятельство, что код LINPACK всегда ссылается на массивы вниз по столбцам, а не от начала до конца строк. Поскольку Фортран хранит массивы, развертывая их по столбцам, это обстоятельство имеет весьма важное значение. Это значит, что при направлении вниз по столбцу массива ссылки к памяти направляются к ее ячейкам последовательно. Таким образом, если программа ссылается на элемент отдельного блока, то следующая ссылка, скорее всего, будет относиться к тому же блоку.

За счет включения в пакет подпрограмм Level-1 BLAS программное обеспечение LINPACK сохраняло частичную независимость от машинной реализации. Почти все вычисления велись за счет вызова Level-1 BLAS. С целью достижения высокой производительности набор Level-1 BLAS мог быть реализован на каждой машине в соответствии с ее особенностями. В примере 14.2 показано, как в LINPACK реализована факторизация.

Пример 14.2. LINPACK вариант (на Фортране 66)

```

subroutine dgefa(a, lda, n, ipvt, info)
integer lda, n, ipvt(1), info
double precision a(lda,1)
double precision t
integer idamax,j,k,kp1,l,nm1
C
C   Гауссово исключение с частичным выбором ведущего элемента
C
info = 0
nm1 = n - 1
if (nm1 .lt. 1) go to 70
do 60 k = 1, nm1
    kp1 = k + 1
C
C   нахождение l = индексу ведущего элемента
    l = idamax(n-k+1,a(k,k),1) + k - 1
    ipvt(k) = l
C
C   нулевой ведущий элемент означает, что этот столбец уже
C   триангулизирован
C
    if (a(l,k) .eq. 0.0d0) go to 40
C

```

```

C      по необходимости - взаимный обмен
C
C      if (l .eq. k) go to 10
C          t = a(l,k)
C          a(l,k) = a(k,k)
C          a(k,k) = t
10    continue
C
C      вычисление множителей
C
C      t = -1.0d0/a(k,k)
C      call dscal(n-k,t,a(k+1,k),1)
C
C      исключение строк с индикацией столбцов
C
C      do 30 j = kp1, n
C          t = a(l,j)
C          if (l .eq. k) go to 20
C              a(l,j) = a(k,j)
C              a(k,j) = t
20    continue
C          call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
30    continue
C          go to 50
40    continue
C          info = k
50    continue
60    continue
70    continue
C          ipvt(n) = n
C          if (a(n,n) .eq. 0.0d0) info = n
C          return
C      end

```

Из набора Level-1 BLAS в подпрограмме DGEFA используются подпрограммы DAXPY, DSCAL и IDAMAX. Основное различие примеров 14.1 и 14.2 (кроме языков программирования и замены индексов цикла) состоит в использовании подпрограммы DAXPY для кодирования внутреннего цикла этого метода.

Предполагалось, что операции BLAS будут осуществлены эффективным, машинно-зависимым способом, подходящим для компьютера, на котором выполняются подпрограммы. На векторном компьютере они могут быть отрансформированы в простые одновекторные операции. Это бы освободило компилятор от оптимизации кода и явного воздействия на критичные по производительности операции.

В определенном смысле тогда красота исходного кода была восстановлена с использованием нового словаря описания алгоритмов: BLAS. Со временем BLAS стал общепринятым стандартом и, скорее всего, был первым пакетом, в котором нашли свое осуществление два ключевых аспекта программного обеспечения: модульность и переносимость. Хотя в наше время это считается само собой разумеющимся, но в ту пору такого еще не было. Получить компактное представление алгоритма и воспользоваться им мог кто угодно, поскольку получаемый в итоге код на Фортране был переносимым.

Большинство алгоритмов линейной алгебры могут быть легко векторизованы. Тем не менее, чтобы получить от этой архитектуры наибольшую выгоду, простой векторизации, как правило, недостаточно. Некоторые векторные компьютеры ограничены наличием единственного маршрута между памятью и векторными регистрами. Если программа осуществляет загрузку векторов из памяти, выполняет ряд арифметических операций, а затем сохраняет результаты, это обстоятельство становится узким местом. Для достижения наивысшей производительности рамки векторизации должны быть расширены, чтобы в дополнение к использованию векторных операций помочь осуществлению совместных конвейерных операций и минимизировать перемещения данных. Переделка алгоритмов в виде их перевода на матрично-векторные операции облегчает векторизированным компиляторам добиться осуществления этих целей.

Из-за того что архитектура компьютеров в конструкции их иерархии памяти стала сложнее, пришло время расширить рамки подпрограмм BLAS с Level-1 до Level-2 и Level-3.

LAPACK DGETRF

Ранее уже упоминалось, что появление на рубеже 70–80 годов векторных машин повлекло за собой разработку других вариантов алгоритмов для решения задач плотной линейной алгебры. Сердцевиной алгоритмов стало векторное умножение матриц. Эти подпрограммы предназначались для получения лучшей производительности по сравнению с теми подпрограммами плотной линейной алгебры LINPACK, которые базировались на уровне Level-1 BLAS. Позже, с появлением на рубеже 80–90 микропроцессоров RISC-типа («убийческих микросов») и других машин с блоками памяти кэш-типа, мы наблюдали разработку алгоритмов для плотной линейной алгебры LAPACK Level-3. Код Level-3 воплотился в основном наборе Level-3 BLAS, который в данном случае представляет собой матричное умножение.

Первоначальная цель проекта LAPACK заключалась в создании библиотеки широкого применения LINPACK, эффективно работающей на векторных и параллельных процессорах с общим пространством памяти. На этих машинах LINPACK работал неэффективно, поскольку его схемы доступа к памяти игнорировали присущую машинам многоуровневую иерархию памяти, из-за чего тратили слишком много времени на перемещение данных вместо того, чтобы выполнять полезные операции с плавающей точкой. LAPACK обошел эту проблему за счет реорганизации алгоритмов на использование блочных матричных операций, таких как матричное умножение, в самых внутренних циклах (см. статью Е. Андерсона (E. Anderson) и Дж. Донгарра (J. Dongarra), указанную в списке дополнительной литературы). Эти блочные операции могут быть оптимизированы под любую архитектуру с учетом ее иерархией памяти, обеспечивая тем самым способ достижения высокой эффективности, который может быть приспособлен к различным современным машинам.

Здесь мы вместо термина «переносимость» воспользовались термином «при-способляемость», поскольку для наилучшей производительности LAPACK

требует, чтобы на каждой машине уже были реализованы хорошо оптимизированные блочные матричные операции. Иными словами, код в точности может быть перенесен, а высокая производительность – не может, если мы ограничим себя лишь исходным кодом на Фортране.

LAPACK с точки зрения функциональности может рассматриваться в качестве наследника LINPACK, несмотря на то что он не всегда использует такие же последовательности вызова функций. В качестве наследника LAPACK был для научного сообщества настоящим приобретением, поскольку он мог сохранять функциональность LINPACK наряду с улучшенным использованием нового аппаратного обеспечения.

В примере 14.3 показано, как в LAPACK решается задача LU-факторизации.

Пример 14.3. Решение в LAPACK задачи факторизации

```
SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
  INTEGER INFO, LDA, M, N
  INTEGER IPIV( * )
  DOUBLE PRECISION A( LDA, * )
  DOUBLE PRECISION ONE
  PARAMETER ( ONE = 1.0D+0 )
  INTEGER I, IINFO, J, JB, NB
  EXTERNAL DGEMM, DGETF2, DLASWP, DTRSM, XERBLA
  INTEGER ILAENV
  EXTERNAL ILAENV
  INTRINSIC MAX, MIN
  INFO = 0
  IF( M.LT.0 ) THEN
    INFO = -1
  ELSE IF( N.LT.0 ) THEN
    INFO = -2
  ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
    INFO = -4
  END IF
  IF( INFO.NE.0 ) THEN
    CALL XERBLA( 'DGETRF', -INFO )
    RETURN
  END IF
  IF( M.EQ.0 .OR. N.EQ.0 ) RETURN
  NB = ILAENV( 1, 'DGETRF', ' ', M, N, -1, -1 )
  IF( NB.LE.1 .OR. NB.GE.MIN( M, N ) ) THEN
    CALL DGETF2( M, N, A, LDA, IPIV, INFO )
  ELSE
    DO 20 J = 1, MIN( M, N ), NB
      JB = MIN( MIN( M, N )-J+1, NB )
*     факторизация диагональных и поддиагональных блоков
*     и тест на точную сингулярность.
      CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
*     Настройка INFO и индексов ведущего элемента.
      IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
      DO 10 I = J, MIN( M, J+JB-1 )
        IPIV( I ) = J - 1 + IPIV( I )
10    CONTINUE
20    CONTINUE
  END IF
END SUBROUTINE DGETRF
```

```

* Применение замен к столбцам 1:J-1.
CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )

*
IF( J+JB.LE.N ) THEN
* Применение замен к столбцам J+JB:N.
CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1, IPIV, 1 )
*
* Вычисление блока строк U.
CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
$      N-J-JB+1, ONE, A( J, J ), LDA, A( J, J+JB ), LDA )
IF( J+JB.LE.M ) THEN
* Обновление конечной подматрицы.
CALL DGEMM( 'No transpose', 'No transpose', M-J-JB+1,
$      N-J-JB+1, JB, -ONE, A( J+JB, J ), LDA,
$      A( J, J+JB ), LDA, ONE, A( J+JB, J+JB ), LDA )
END IF
END IF
20 CONTINUE
END IF
RETURN
end

```

Основная часть вычислительной работы алгоритма из примера 14.3 содержится в трех подпрограммах:

DGEMM

Матрично-матричное умножение.

DTRSM

Решение уравнения треугольной матрицы с несколькими правыми сторонами.

DGETF2

Внеблочная LU-факторизация для операций в пределах столбца блока.

Одним из ключевых параметров алгоритма является размер блока, названный здесь NB. Если NB слишком маленький или слишком большой, это выражается в низкой производительности, что выводит на первый план значимость функции ILAENV, для которой предусматривалась замена стандартной реализации на реализацию поставщика, формирующую машинно-зависимые параметры после инсталляции библиотеки LAPACK. В любой заданной точке алгоритма NB строк или столбцов экспонируется на хорошо оптимизированный уровень Level-3 BLAS. Если NB равен единице, алгоритм по производительности и схемам доступа к памяти эквивалентен версии LINPACK.

Матрично-матричные операции предоставляют надлежащий уровень модульности для хорошей производительности и приспособляемости в широком диапазоне компьютерных архитектур, включая параллельные системы, обладающие иерархией памяти. Улучшенная производительность прежде всего достигается за счет расширенных возможностей многократного использования данных. Существует множество способов достижения этого многократного использования данных, чтобы уменьшить в памяти информационный обмен и увеличить отношение операций с плавающей точкой к перемещению данных по уровням иерархии памяти. Такое улучшение на компьютерах с современной архитектурой может увеличить производительность от трех до десяти раз.

Единого мнения о продуктивности чтения и записи кода LAPACK еще не сложилось: вопрос в том, насколько сложно генерировать код из его математического описания? Использование векторной системы записи в LINPACK, возможно, выглядит более естественно, чем матричные формулировки в LAPACK. Описывающие алгоритмы математические формулы в противоположность смешанной матрично-векторной нотации обычно имеют более сложное представление, если используются только одни матрицы.

Рекурсивное использование LU

Установка параметров размера блока для LU в LAPACK на первый взгляд может показаться тривиальной задачей. Но на практике это требует множества настроек под различную точность и размеры матрицы. Многие пользователи останавливаются на том, что оставляют установки без изменений, даже если настройка была произведена только при инсталляции. Проблема усугубляется тем, что не только одна, но многие подпрограммы LAPACK используют параметр разделения на блоки.

Другой проблемой имеющейся в LAPACK формулировки LU является факторизация высоких и узких панелей столбцов, выполняемая подпрограммой DGETF2. Она использует уровень Level-1 BLAS и снискала себе славу узкого места для процессоров, которые с 90-х годов стали намного быстрее, но не увеличили соответствующим образом пропускную способность памяти.

Решение пришло с совершенно неожиданной стороны: рекурсии, использующей принцип «разделяй и властвуй». Вместо циклов LAPACK новый рекурсивный алгоритм LU разбивает работу на две половины, проводит факторизацию левой части матрицы, обновляет оставшуюся часть матрицы и проводит факторизацию правой части. Использование Level-1 BLAS сокращается до приемлемого минимума, и при большинстве вызовов Level-3 BLAS обрабатываются более крупные части матрицы, чем в алгоритме LAPACK. И разумеется, теперь настройка на размер блока уже не требуется.

Для рекурсивного LU нужно было использовать Фортран 90, в котором были реализованы первые фортрановские стандарты, разрешающие применение рекурсивных подпрограмм. Побочным эффектом использования Фортрана 90 было увеличение значимости параметра LDA, начальной A. Это позволяет использовать подпрограмму более гибко, а также настраивать производительность для тех случаев, когда размерность матрицы m вызывала бы конфликты в группе блоков памяти, способные значительно уменьшить доступную полосу ее пропускания.

Компилятор Фортрана 90 использует параметр LDA, чтобы избежать копирования данных в непрерывный буфер при вызове внешних подпрограмм, в частности какой-нибудь подпрограммы BLAS. Без LDA компилятор должен предполагать наихудший сценарий развития событий, когда ввод матрицы A не является непрерывным и нуждается в копировании во временный непрерывный буфер, поэтому вызов BLAS не заканчивается доступом к неограниченной

памяти. С использованием LDA компилятор передает BLAS указатели массива, не делая никаких копий.

В примере 14.4 показана рекурсивная LU-факторизация.

Пример 14.4. Рекурсивный вариант (на Фортране 90)

```

recursive subroutine rdgetrf(m, n, a, lda, ipiv, info)
implicit none

integer, intent(in) :: m, n, lda
double precision, intent(inout) :: a(lda,:)
integer, intent(out) :: ipiv(*)
integer, intent(out) :: info

integer :: mn, nleft, nright, i
double precision :: tmp

double precision :: pone, negone, zero
parameter (pone=1.0d0)
parameter (negone=-1.0d0)
parameter (zero=0.0d0)

intrinsic min

integer idamax
external dgemm, dtrsm, dlaswp, idamax, dscal

mn = min(m, n)

if (mn .gt. 1) then
    nleft = mn / 2
    nright = n - nleft

    call rdgetrf(m, nleft, a, lda, ipiv, info)

    if (info .ne. 0) return
    call dlaswp(nright, a(1, nleft+1), lda, 1, nleft, ipiv, 1)

    call dtrsm('L', 'L', 'N', 'U', nleft, nright, pone, a, lda,
$      a(1, nleft+1), lda)
    call dgemm('N', 'N', mn-nleft, nright, nleft, negone,
$      a(nleft+1, 1), lda, a(1, nleft+1), lda, pone,
$      a(nleft+1, nleft+1), lda)

    call rdgetrf(m - nleft, nright, a(nleft+1, nleft+1), lda,
$      ipiv(nleft+1), info)
    if (info .ne. 0) then
        info = info + nleft
        return
    end if

    do i = nleft+1, m
        ipiv(i) = ipiv(i) + nleft
    end do
end if

```

```

call dlaswp(nleft, a, lda, nleft+l, mn, ipiv, 1)

else if (mn .eq. 1) then
  i = idamax(m, a, 1)
  ipiv(1) = i
  tmp = a(i, 1)

  if (tmp .ne. zero .and. tmp .ne. -zero) then
    call dscal(m, pone/tmp, a, 1)
    a(i,1) = a(1,1)
    a(1,1) = tmp
  else
    info = 1
  end if

end if

return
end

```

В рекурсивном варианте есть определенная степень утонченности. В подпрограмме отсутствуют какие-либо циклы. Вместо них алгоритм управляетя рекурсивной природой метода (см. статью Ф. Г. Густавсона (F. G. Gustavson), указанную в списке дополнительной литературы).

Рекурсивный алгоритм LU содержит четыре основных шага, показанных на рис. 14.2.

1. Разбиение матрицы на два прямоугольника ($m * n/2$); если оказывается, что левая часть представляет собой одиночный столбец, она масштабируется за счет реверсирования оси и возвращается.
2. Применение к левой части LU-алгоритма.
3. Применение преобразования к правой части (решения уравнения треугольной матрицы $A_{12} = L^{-1}A_{12}$ и перемножение матриц $A_{22} = A_{22} - A_{21} * A_{12}$).
4. Применение LU-алгоритма к правой части.

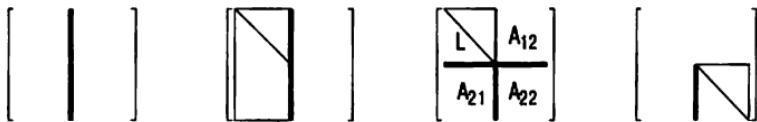


Рис. 14.2. Рекурсивная LU-факторизация

Основная часть работы выполняется при перемножении матриц, применяемом к последовательности матриц размера $n/2$, $n/4$, $n/8$ и т. д. Реализация, показанная в примере 14.4, демонстрирует примерно 10-процентное приращение производительности по сравнению с реализацией LAPACK, показанной в примере 14.3.

В известном смысле любое предыдущее воспроизведение LU-алгоритма может рассматриваться в понятиях изящества кода как шаг назад. Но рекурсия, выполненная по принципу «разделяй и властвуй», была огромным скачком вперед (даже несмотря на довольно скромное приращение производительности).

Теперь рекурсивный алгоритм для факторизации матрицы может преподаваться студентам наряду с другими рекурсивными алгоритмами, такими как различные виды методов сортировки.

За счет изменения всего лишь размеров матричных частей появляется возможность добиться такой схемы доступа к памяти, которая используется LINPACK или в LAPACK. Установка значения $nleft$ равным 1 придает коду способность работы с векторами как в LINPACK, а установка значения $nleft$ равным $NB > 1$ придает ему рабочие свойства, подобные блочному коду LAPACK. В обоих случаях исходная рекурсия деградирует от поддержки принципа «разделяй и властвуй» до самого скромного функционирования. Поведение таких вариантов рекурсивного алгоритма может служить учебным пособием наряду с алгоритмом Quicksort, используемым с различными схемами разделения сортируемого массива.

В завершение в качестве упражнения мы предоставляем читателю возможность предпринять попытку сымитировать рекурсивный код без использования рекурсии и без явного управления стеком вызова рекурсий, то есть решить важную проблему для того случая, когда компилятор Фортрана не в состоянии обрабатывать рекурсивные функции или подпрограммы.

ScaLAPACK PDGETRF

LAPACK был спроектирован в расчете на высокую эффективность при работе на векторных процессорах, высокопроизводительных «суперскалярных» рабочих станциях и мультипроцессорах с общим пространством памяти. LAPACK может также удовлетворительно работать на всех типах скалярных машин (персональных компьютерах, рабочих станциях и универсальных вычислительных машинах). Но в существующей на данный момент форме LAPACK вряд ли сможет выдать хорошую производительность на машинах с другим типом параллельных архитектур, например, на машинах с одним потоком команд и многопоточными данными –Single Instruction Multiple Data (SIMD) или машинах с многопоточными командами и многопоточными данными с распределенным пространством памяти – Multiple Instruction Multiple Data (MIMD).

Для адаптации LAPACK к этим новым архитектурам были предприняты усилия по созданию ScaLAPACK. Создание программной библиотеки ScaLAPACK позволило расширить действие библиотеки LAPACK на работу с расширяемыми параллельными MIMD-компьютерами с распределенным пространством памяти. Иерархия памяти таких машин вдобавок к иерархии регистров, кэш-памяти и локальной памяти каждого процессора включает еще и внепроцессорную память других процессоров.

В целях минимизации частоты перемещений данных между различными уровнями иерархии памяти подпрограммы ScaLAPACK, так же как и подпрограммы LAPACK, основаны на алгоритмах блочного разбиения. Основные стандартные блоки библиотеки ScaLAPACK представляют собой версии Level-2 и Level-3 BLAS, приспособленные для работы с распределенной памятью, и наборы подпрограмм Basic Linear Algebra Communication Subprograms (BLACS)

для решения задач по передаче информации, часто востребуемых при параллельных решениях задач линейной алгебры.

Весь межпроцессорный информационный обмен в подпрограммах ScalAPACK происходит в пределах распределенных наборов BLAS и BLACS, поэтому исходный код верхнего программного уровня ScalAPACK очень похож на исходный код LAPACK.

В примере 14.5 показано, как задача LU-факторизации решена в ScalAPACK.

Пример 14.5. Вариант ScalAPACK (на Фортране 90)

```

SUBROUTINE PDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
  INTEGER             BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_
  $                  LLD_, MB_, M_, NB_, N_, RSRC_
  $                  ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DTYPE_ = 1,
  $                  CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
  $                  RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
  DOUBLE PRECISION   ONE
  PARAMETER          ( ONE = 1.0D+0 )
  CHARACTER          COLBTOP, COLCTOP, ROWBTOP
  INTEGER             ICOFF, ICTXT, IINFO, IN, IROFF, J, JB, JN,
  $                  MN, MYCOL, MYROW, NPCOL, NPROW
  $                  IDUM1( 1 ), IDUM2( 1 )
  EXTERNAL            BLACS_GRIDINFO, CHK1MAT, IGAMN2D, PCHK1MAT, PB_TOPGET,
  $                  PB_TOPSET, PDGEMM, PDGETF2, PDLASWP, PDTRSR, PXERBLA
  INTEGER             ICEIL
  EXTERNAL            ICEIL
  INTRINSIC           MIN, MOD
*
* Получение параметров области вычислений
* ICTXT = DESCA( CTXT_ )
* CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
*
* Тестирование входных параметров
* INFO = 0
* IF( NPROW.EQ.-1 ) THEN
*     INFO = -(600+CTXT_)
* ELSE
*     CALL CHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, INFO )
*     IF( INFO.EQ.0 ) THEN
*         IROFF = MOD( IA-1, DESCA( MB_ ) )
*         ICOFF = MOD( JA-1, DESCA( NB_ ) )
*         IF( IROFF.NE.0 ) THEN
*             INFO = -4
*         ELSE IF( ICOFF.NE.0 ) THEN
*             INFO = -5
*         ELSE IF( DESCA( MB_ ).NE.DESCA( NB_ ) ) THEN
*             INFO = -(600+NB_)
*         END IF
*     END IF
*     CALL PCHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, 0,
*                   IDUM1, IDUM2, INFO )
* END IF
* IF( INFO.NE.0 ) THEN
*     CALL PXERBLA( ICTXT, 'PDGETRF', -INFO )
* RETURN

```

```

END IF
IF( DESCA( M_ ).EQ.1 ) THEN
    IPIV( 1 ) = 1
    RETURN
ELSE IF( M.EQ.0 .OR. N.EQ.0 ) THEN
    RETURN
END IF
*
* Топология кольцевого разбиения для обмена информацией на
* время обработки строк
CALL PB_TOPGET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
CALL PB_TOPGET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
CALL PB_TOPGET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', 'S-ring' )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', ' ' )
CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', ' ' )
*
* Отдельная обработка первого блока столбцов
MN = MIN( M, N )
IN = MIN( ICEIL( IA, DESCA( MB_ ) )*DESCA( MB_ ), IA+M-1 )
JN = MIN( ICEIL( JA, DESCA( NB_ ) )*DESCA( NB_ ), JA+MN-1 )
JB = JN - JA + 1
*
* Факторизация диагональных и поддиагональных блоков и тест
* на точную сингулярность.
CALL PDGETF2( M, JB, A, IA, JA, DESCA, IPIV, INFO )
IF( JB+1.LE.N ) THEN
    *
    * Применение замен к столбцам JN+1:JA+N-1.
    CALL PDLASWP('Forward', 'Rows', N-JB, A, IA, JN+1,
$ DESCA, IA, IN, IPIV )
    *
    * Вычисление блока строк U.
    CALL PDTRS( 'Left', 'Lower', 'No transpose', 'Unit',
$           JB, N-JB, ONE, A, IA, JA, DESCA, A, IA,
$           JN+1, DESCA )
    *
    IF( JB+1.LE.M ) THEN
        *
        * Обновление конечной подматрицы.
        CALL PDGEMM( 'No transpose', 'No transpose',
$                   M-JB, N-JB, JB,-ONE, A, IN+1,
$                   JA, DESCA, A, IA, JN+1, DESCA,
$                   ONE, A, IN+1, JN+1, DESCA )
    END IF
END IF
*
* Последовательный перебор оставшихся блоков столбцов.
DO 10 J = JN+1, JA+MN-1, DESCA( NB_ )
JB = MIN( MN-J+JA, DESCA( NB_ ) )
I = IA + J - JA
*
* Факторизация диагональных и поддиагональных блоков и
* тест на точную сингулярность.
CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )
IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - JA
*
* Применение замен к столбцам JA:J-JA.
CALL PDLASWP('Forward', 'Rowwise', J-JA, A, IA, JA,
$ DESCA, I,I+JB-1, IPIV)

```

```

*           IF( J-JA+JB+1.LE.N ) THEN
*               Применение замен к столбцам J+JB:JA+N-1.
*               CALL PDLASWP( 'Forward', 'Rowwise', N-J-JB+JA, A,
*                               IA, J+JB, DESCA, I, I+JB-1, IPIV )
*               Вычисление блока строк U.
*               CALL PDTRSM( 'Left', 'Lower', 'No transpose',
*                             'Unit', JB, N-J-JB+JA, ONE, A, I, J,
*                             DESCA, A, I, J+JB, DESCA )
*           IF( J-JA+JB+1.LE.M ) THEN
*               Обновление конечной подматрицы.
*               CALL PDGEMM( 'No transpose', 'No transpose',
*                             M-J-JB+JA, N-J-JB+JA, JB, -ONE,
*                             A, I+JB, J, DESCA, A, I, J+JB,
*                             DESCA, ONE, A, I+JB, J+JB, DESCA )
*           END IF
*       END IF
10 CONTINUE
IF( INFO.EQ.0 ) INFO = MN + 1
CALL IGAMN2D( ICTXT, 'Rowwise', ' ', 1, 1, INFO, 1,
$                 IDUM1, IDUM2, -1,-1, MYCOL)
IF( INFO.EQ.MN+1 ) INFO = 0
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
RETURN
END

```

Чтобы упростить конструкцию ScalAPACK, и учитывая, что пакет BLAS за пределами LAPACK зарекомендовал себя весьма полезным инструментом, мы решили разработать параллельный (Parallel) BLAS, или PBLAS (который описан в статье Чои (Choi) и др., представленной в списке дополнительной литературы), чей интерфейс, насколько возможно, похож на интерфейс BLAS. Это решение дало возможность коду ScalAPACK стать весьма похожим, а иногда и почти идентичным аналогичному коду LAPACK.

Нашей целью было обеспечить работу PBLAS по стандартам распределенной памяти, так же как в BLAS обеспечивалась работа с общим пространством памяти. Тем самым была бы упрощена и поощрена разработка высокоэффективного и переносимого параллельного числового программного обеспечения, а производителям был бы предоставлен весьма скромный набор подпрограмм, требующих оптимизации. Положительное восприятие PBLAS требует достижения разумного компромисса между конкурирующими целями функциональности и простоты.

PBLAS работает с матрицами, распределенными по схеме двумерных циклических блоков. Поскольку для такой схемы требуется использование множества параметров, полностью описывающих распределенную матрицу, мы выбрали более выраженный объектно-ориентированный подход и инкапсулировали эти параметры в целочисленном массиве, названным описателем массива — *array descriptor*. Описатель массива включает следующие элементы.

- Тип описателя.
- BLACS-контекст (виртуальное пространство для сообщений, которые создаются во избежание коллизий между логически отличающихся сообщений).

- Количество строк распределенной матрицы.
- Количество столбцов распределенной матрицы.
- Размер блока строк.
- Размер блока столбцов.
- Стока процесса, по которой распределяется первая строка матрицы.
- Столбец процесса, по которому распределяется первый столбец матрицы.
- Начальная размерность локального массива, сохраняющего локальные блоки.

Благодаря использованию этого описателя, вызов подпрограммы PBLAS очень похож на вызов соответствующей подпрограммы BLAS:

```
CALL DGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
            A( IA, JA ), LDA,
            B( IB, JB ), LDB, BETA,
            C( IC, JC ), LDC )
```

```
CALL PDGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
              A, IA, JA, DESC_A,
              B, JB, DESC_B, BETA,
              C, IC, JC, DESC_C )
```

DGEMM вычисляет $C = BETA * C + ALPHA * op(A) * op(B)$, где $op(A)$ — это или A , или ее транспозиция, в зависимости от $TRANS_A$, $op(B)$ — аналогично, $op(A)$ имеет размер M на K , а $op(B)$ — K на N . PDGEMM делает то же самое, за исключением способа определения подматрицы. К примеру, для передачи DGEMM подматрицы, начинающейся с $A(IA,JA)$, фактический параметр, соответствующий формальному аргументу A , — это просто $A(IA,JA)$. В свою очередь PDGEMM для извлечения соответствующей подматрицы требует понимания глобальной схемы хранения A , поэтому IA и JA должны быть переданы отдельно.

Описателем матрицы A служит массив $DESC_A$. Параметры, описывающие матричные операнды B и C , аналогичны тем, которые описывают A . В реальной объектно-ориентированной среде матрица и $DESC_A$ были бы тождественными понятиями. Но для этого нужна языковая поддержка и уступки в вопросах переносимости.

Использование передачи сообщений и масштабируемых алгоритмов библиотеки ScaLAPACK позволяет при оснащении машин дополнительным количеством процессоров производить факторизацию матрицы произвольно увеличивающихся размеров. По своей конструкции библиотека больше занимается вычислением, чем передачей информации, поэтому данные большей частью проходят обработку в локальных пространствах и лишь время от времени перемещаются по сети взаимодействия. Но с количеством и типами сообщений, которыми процессоры обмениваются друг с другом, порой не так-то легко справиться. Контекст, связанный с каждой распределенной матрицей, позволяет реализации библиотеки для передачи сообщений использовать обособленные «области». Использование отдельных коммуникационных контекстов для различных библиотек (или для различных библиотечных вызовов), таких как PBLAS, позволяет изолировать внутренний библиотечный процесс передачи

информации от внешнего. Если в перечне аргументов подпрограммы PBLAS присутствует более одного описателя массива, индивидуальные контекстные входы BLACS должны равняться их числу. Иными словами, PBLAS не выполняет «межконтекстных» операций.

С точки зрения производительности ScaLAPACK сделал по сравнению с LAPACK то же самое, что LAPACK сделал по сравнению с LINPACK: расширил диапазон аппаратных средств, на которых LU-факторизация (и другой код) может работать эффективно. С точки зрения изящества изменения, произошедшие при создании ScaLAPACK, были более радикальными: те же самые математические операции теперь требуют большего количества трудоемкой работы. И пользователи, и создатели библиотеки теперь вынуждены заняться непосредственным управлением всеми хитросплетениями хранения данных, поскольку вопрос их размещения стал основным фактором производительности. В жертву была принесена легкость чтения кода, несмотря на те усилия, которые были потрачены на разбиение кода на модули, в соответствии с лучшим имеющимся на сегодняшний день опытом разработки программного обеспечения.

Многопоточная обработка для многоядерных систем

Появление многоядерных микропроцессоров привело к фундаментальным изменениям в способах производства программного обеспечения. Задачи плотной линейной алгебры тоже не стали исключением. Положительный момент заключается в том, что LU-факторизация LAPACK работает и на многоядерных системах и может даже продемонстрировать небольшой прирост производительности, если будет использован многопоточный BLAS. На техническом языке это называется моделью вычислений fork-join (ветвление-объединение): каждый вызов BLAS (из единого основного потока) приводит к разветвлению на подходящее число потоков, которые выполняют работу на каждом из ядер, а затем объединяют ее в основном потоке вычислений. Модель ветвления-объединения предполагает в каждой операции объединения определенную точку синхронизации.

Отрицательным моментом является то, что именно на небольших многоядерных компьютерах, не имеющих систем памяти, доступных на симметричных мультипроцессорных системах, алгоритм ветвления-объединения LAPACK существенно ухудшает масштабируемость. Врожденный порок масштабируемости заключается в трудностях синхронизации в модели ветвления-объединения (самые главные вычисления, занимающие критический фрагмент кода, разрешено выполнять только единственному потоку, оставляя другие потоки в режиме ожидания), это приводит к выполнению в режиме блокировки шагов (lock-step execution) и мешает скрытию по своей природе последовательных частей кода за теми, которые выполняются параллельно с ними. Иными словами, потоки вынуждены выполнять ту же самую операцию на различных данных. Если для некоторых потоков недостаточно данных, они будут находиться в вынужденном простое, ожидая остальные потоки, производящие полезную работу над своими данными. Очевидно, в качестве следующего LU-алгоритма нужен

такой, который позволит потокам оставаться занятыми все время, дав им возможность выполнять различные операции в период выполнения какой-то части программы. Многопотоковая версия алгоритма распознает выполнение в алгоритме так называемого *критического пути*: части кода, чье выполнение зависит от предыдущих вычислений и может блокировать дальнейшее выполнение алгоритма. Имеющийся в LAPACK LU не обрабатывает эту критическую часть кода каким-либо особым способом: подпрограмма DGETF2 вызывается для одного потока и не позволяет широко использовать распараллеливание вычислений даже на уровне BLAS. В то время как один поток вызывает эту подпрограмму, другие находятся в ожидании. А так как производительность DGETF2 связана с пропускной способностью памяти (а не со скоростью процессора), это узкое место только усугубит проблемы масштабируемости с появлением систем с большим количеством ядер.

Многопоточная версия алгоритма берется за решение этой проблемы «в лоб», вводя понятие предвидения: заблаговременного вычисления во избежание потенциального застоя в ходе вычислений. Разумеется, это требует дополнительной синхронизации и регистрации использования системных ресурсов, которые отсутствуют в предыдущих версиях — выбора компромисса между сложностью кода и производительностью. Другим аспектом многопоточного кода является использование рекурсии в панельной факторизации. Оказалось, что использование рекурсии может дать даже больший прирост производительности для высоких панельных матриц, чем для квадратных.

В примере 14.6 показана факторизация, подходящая для многопоточного выполнения.

Пример 14.6. Факторизация для многопоточного выполнения (на Си)

```
void SMP_dgetrf(int n, double *a, int lda, int *ipiv, int pw,
                int tid, int tsize, int *pready.ptm *mtx, ptc *cnd) {
    int pcnt, pfctr, ufrom, uto, ifrom, p;
    double *pa = a, *pl, *pf, *lp;

    pcnt = n / pw; /* количество панелей */
    pfctr = tid + (tid ? 0 : tsize); /* первая панель, которая подлежит
                                       факторизации сразу же после факторизации самой первой
                                       панели (с нулевым номером) */

    /* это указатель на последнюю панель */
    lp = a + (size_t)(n - pw) * (size_t)lda;

    /* для каждой панели (используемой в качестве источника обновления) */
    for (ufrom = 0; ufrom < pcnt; ufrom++) {
        pa += (size_t)pw *
              (size_t)(lda + 1));
        p = ufrom * pw; /* номер столбца */

        /* Если панель, использованная для обновления, еще не была
           факторизирована; 'ipiv' не принимается во внимание, но это относится
           к возможному уклонению от обращений к 'pready'*/
        if (! ipiv[p + pw - 1] || ! pready[ufrom]) {
            if (ufrom % tsize == tid) { /* если это панель этого потока */
                /* здесь идет факторизация панели */
            }
        }
    }
}
```

```

pfactor( n - p, pw, pa, lda, ipiv + p, pready, ufrom, mtx, cnd );
} else if (ufrom < pcnt - 1) { /* если эта панель не последняя */
    LOCK( mtx );
    while (! pready[ufrom]) { WAIT( cnd, mtx ); }
    UNLOCK( mtx );
}

/* Для каждой обновляемой панели */
for (uto = first_panel_to_update( ufrom, tid, tsize ); uto < pcnt;
    uto += tsize) {
/* Если это все еще панели для факторизации этим потоком, и предыдущая
панель была факторизирована; проверка на 'ipiv' может быть пропущена.
но она оставлена для уменьшения количества обращений к 'pready' */
if (pfctr < pcnt && ipiv[pfctr * pw - 1] && pready[pfctr - 1]) {
    /* для каждой панели, которая должна (все еще) обновить
панель 'pfctr' */
    for (ifrom = ufrom + (uto > pfctr ? 1 : 0); ifrom < pfctr; ifrom++) {
        p = ifrom * pw;
        pl = a + (size_t)p * (size_t)(lda + 1);
        pf = pl + (size_t)(pfctr - ifrom) * (size_t)pw * (size_t)lda;
        pupdate( n - p, pw, pl, pf, lda, p, ipiv, lp );
    }
    p = pfctr * pw;
    pl = a + (size_t)p * (size_t)(lda + 1);
    pfactor( n - p, pw, pl, lda, ipiv + p, pready, pfctr, mtx, cnd );
    pfctr += tsize; /* перемещение к следующей панели этого потока */
}

/* если панель 'uto' не была факторизирована (если была, то она,
безусловно, была обновлена, и обновление не требуется) */
if (uto > pfctr || ! ipiv[uto * pw]) {
    p = ufrom * pw;
    pf = pa + (size_t)(uto - ufrom) * (size_t)pw * (size_t)lda;
    pupdate( n - p, pw, pa, pf, lda, p, ipiv, lp );
}
}

```

Для каждого потока используется один и тот же алгоритм (парадигма SIMD), и данные матрицы делятся на части между потоками в циклическом режиме, используя панели по ρ столбцов в каждой (за исключением, может быть, последней). Параметр ρ соответствует имеющемуся в LAPACK параметру разбиения на блоки NB. Разница заключается в логическом распределении панелей (блоков столбцов) по потокам. (Физически все панели одинаково доступны, поскольку код работает в режиме общего использования памяти.) Преимущества от разбиения на блоки по потокам те же самые, что были в LAPACK: улучшенное многократное использование кэш-памяти и меньшая нагрузка на шину памяти.

Назначение части матрицы потоку поначалу кажется искусственным требованием, но это упрощает код и учет структур данных; и, что наиболее важно, обеспечивает лучшую доступность памяти. Оказывается, что многоядерные кристаллы не обладают симметрией в отношении пропускной способности памяти, поэтому минимизация количества перераспределений страниц памяти между ядрами непосредственно влияет на повышение производительности.

Стандартные компоненты LU-факторизации представлены функциями `rfactor()` и `update()`. Как можно было бы предположить, первая из них проводит факторизацию панели, а вторая — обновляет панель, используя одну из ранее факторизованных панелей.

Основной цикл заставляет каждый поток выполнять циклическую обработку каждой панели по очереди. Если необходимо, панель проходит факторизацию владеющим ею потоком, пока другие панели находятся в ожидании (если случается, что эта панель им нужна для проводимых ими обновлений).

Логика предвидения находится внутри вложенного цикла (предваряемого комментарием Для каждой обновляемой панели), который заменяет DGEMM или PDGEMM из предыдущего алгоритма. Перед тем как каждый поток обновляет одну из своих панелей, он проверяет, не проводил ли он факторизацию своей первой неотфакторизованной панели. Это минимизирует число случаев, когда потоки вынуждены простоять в ожидании, поскольку каждый поток постоянно пытается устранить потенциально узкое место. Как это произошло с ScaLAPACK, многопоточная версия сокращает унаследованную от версии LAPACK элегантность. Так же и в том же духе главным виновником этого выступает производительность: код LAPACK не будет эффективно работать на машинах с постоянно увеличивающимся количеством ядер. Явное управление рабочими потоками на уровне LAPACK, а не на уровне BLAS является решающим моментом: параллелизм не может быть инкапсулирован в библиотечном вызове. Единственным утешением служит то, что код не настолько сложен, как тот, что используется в ScaLAPACK, и для эффективного BLAS все еще может быть найдено неплохое применение.

Несколько слов об анализе ошибок и итоговом количестве операций

Ключевым аспектом всех реализаций, присутствующих в этой главе, являются их числовые свойства.

Вполне допустимо воздержаться от элегантности в пользу повышения производительности. Но числовая стабильность имеет жизненно важное значение и не может быть принесена в жертву, поскольку она представляет собой врожденную часть корректности алгоритма. При всей серьезности этих соображений есть некоторые утешения, требующие понимания. Кого-то из читателей может удивить тот факт, что все представленные алгоритмы тождественны, даже при том, что практически невозможно заставить каждый фрагмент кода выдавать точно такой же результат на выходе для совершенно одинаковых входных данных.

Когда дело доходит до повторяемости результатов, то причуды представления чисел с плавающей точкой должны быть четко зафиксированы границами ошибки. Один из способов выражения числового «запаса прочности» предыдущих алгоритмов представлен следующей формулой:

$$\|r\| / \|A\| \leq \epsilon \leq \|A^{-1}\| \|r\|$$

Где ошибка $e = x - y$ представляет собой разницу между вычисленным решением y и правильным решением x , а $r = Ay - b$ — это так называемый «остаток». В общем, предыдущая формула говорит о том, что размер ошибки (параллельные полосы, окружающие значение, показывают норму — меру абсолютного значения) настолько мал, насколько это гарантировано особенностью матрицы A . Поэтому если матрица в числовом смысле близка к сингулярной (некоторые записи настолько малы, что их можно рассматривать как нулевые), алгоритм не даст точного ответа. Но в противном случае можно ожидать относительно хорошее качество результата.

Другое свойство, присущее всем версиям, — это итоговое количество операций: все они выполняют $2/3n^3$ умножений чисел с плавающей точкой и (или) сложений. Их отличает порядок выполнения этих операций. Есть алгоритмы, в которых увеличивается объем работы с числами с плавающей точкой, чтобы сэкономить на перемещениях данных в памяти или на их передаче по сети (особенно для параллельных алгоритмов для систем с распределенной памятью). Но поскольку показанные в этой главе алгоритмы имеют одинаковое количество операций, то вполне правомерно проводить сравнение их производительности. Скорость вычислений (количество операций над числами с плавающей точкой в секунду) может быть использована вместо времени, затрачиваемого на решение задачи, при условии одинакового размера матрицы. Но в некоторых случаях, когда размеры матриц различаются, лучше проводить сравнение скорости вычисления, поскольку оно позволяет сравнивать алгоритмы. Например, последовательный алгоритм на одиночном процессоре может непосредственно сравниваться с параллельным алгоритмом, работающим в большом кластере с матрицей значительно большего размера.

Дальнейшее направление исследований

В этой главе мы рассмотрели развитие конструкции простого, но важного для компьютерной науки алгоритма. Изменения, произошедшие за последние 30 лет, были нужны для того, чтобы следовать за прогрессом компьютерной архитектуры. В некоторых случаях эти изменения были простыми, вроде перестановки порядка следования циклов. В других случаях, как с введением рекурсии и предвидений, они были довольно сложными. Тем не менее в каждом случае способность кода эффективно использовать иерархию памяти является ключом к высокой производительности как на одиночном процессоре, так и на системах с общей и распределенной памятью.

Суть проблемы заключается в значительном увеличении сложности, с которой приходилось и приходится сталкиваться разработчикам программного обеспечения. В настоящее время уже получили широкое распространение двухядерные машины, и, как ожидается, с каждым новым поколением процессоров число ядер будет удваиваться. Но вопреки предположениям старой модели, программисты не смогут рассматривать эти ядра независимо (то есть мультиядерность — это *не* «новая SMP»), поскольку характер их совместного использования ресурсов кристалла отличается от действий отдельных процессоров.

Эта ситуация осложняется еще и другими нестандартными компонентами, которые, как ожидается, будут развернуты в следующих архитектурах, включая смешение различных типов ядер, аппаратные ускорители и системы памяти.

В конце концов, распространение сильно отличающихся друг от друга конструкторских идей показывает, что вопрос о том, как найти наилучшее сочетание всех этих новых ресурсов и компонентов является в значительной степени нерешенным. В обобщенном виде эти изменения рисуют картину будущего, где программистам придется преодолевать намного более сложные и замысловатые проблемы конструирования программного обеспечения, чем те, что были в прошлом, чтобы воспользоваться преимуществами еще большего распараллеливания и более высокой вычислительной мощности, которые им будут предложены новыми архитектурами.

Во всем этом плохо то, что никакой ныне существующий код в один прекрасный момент уже не сможет работать эффективно. Но хорошо, что мы изучили различные способы придания определенной формы исходному простому воспроизведению алгоритма, чтобы можно было ответить на неизменно возрастающие вызовы конструкций аппаратного обеспечения.

Дополнительная литература

- «LINPACK User's Guide», J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, SIAM: Philadelphia, 1979.
- «LAPACK Users' Guide», Third Edition, E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, SIAM: Philadelphia, 1999.
- «ScaLAPACK Users' Guide», L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, SIAM Publications, Philadelphia, 1997.
- «Basic Linear Algebra Subprograms for FORTRAN usage», C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, ACM Trans. Math. Soft., vol. 5, 1979, pp. 308–323.
- «An extended set of FORTRAN Basic Linear Algebra Subprograms», J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Trans. Math. Soft., vol. 14, 1988, pp. 1–17.
- «A set of Level 3 Basic Linear Algebra Subprograms», J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, ACM Trans. Math. Soft., vol. 16, 1990, pp. 1–17.
- «Implementation Guide for LAPACK», E. Anderson and J. Dongarra, UT-CS-90-101, Apr. 1990.
- «A Proposal for a Set of Parallel Basic Linear Algebra Subprograms», J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, UT-CS-95-292, May 1995.
- «LAPACK Working Note 37: Two Dimensional Basic Linear Algebra Communication Subprograms», J. Dongarra and R. A. van de Geijn, University of Tennessee Computer Science Technical Report, UT-CS-91-138, Oct. 1991.

- «Matrix computations with Fortran and paging», Cleve B. Moler, Communications of the ACM, 15(4), 1972, pp. 268–270.
- «LAPACK Working Note 19: Evaluating Block Algorithm Variants in LAPACK», E. Anderson and J. Dongarra, University of Tennessee Computer Science Technical Report, UT-CS-90-103, Apr. 1990.
- «Recursion leads to automatic variable blocking for dense linear-algebra algorithms», Gustavson, F. G. IBM J. Res. Dev. Vol. 41, № 6 (Nov. 1997), pp. 737–756.

15 Долговременные выгоды от красивой конструкции

Адам Колава (*Adam Kolawa*)

Некоторые алгоритмы решения, казалось бы, простых и понятных математических уравнений на поверку оказываются невероятно трудными в реализации. Например, проблемы с округлением могут снизить точность результата, решения некоторых математических уравнений могут привести к превышению допустимого для данной системы диапазона значений чисел с плавающей точкой, а на некоторые составленные «в лоб» алгоритмы (особенно классический алгоритм преобразования Фурье) тратится слишком много времени. Кроме того, для различных наборов данных существуют разные алгоритмы, которые с ними справляются лучше других. Следовательно, красивый программный код и красивая математика — это не обязательно одно и то же.

Программисты, написавшие код для математической библиотеки лаборатории ЦЕРН, понимали разницу между математическими уравнениями и компьютерными решениями, то есть разницу между теорией и практикой. В этой главе я проведу анализ красоты, присущей некоторым стратегиям программирования, использовавшимся ими для преодоления этого разрыва.

В чем, по-моему, заключается красота программного кода

Мое понятие красоты кода базируется на уверенности в том, что он в первую очередь должен быть работоспособен. Иными словами, код должен точно и эффективно выполнять задачу, для решения которой он создавался, причем делать это так, чтобы не было никаких неоднозначных линий его поведения.

Я вижу красоту в том коде, которому могу доверять, — в коде, о котором я с уверенностью могу сказать, что он выдаст точные результаты, вполне приемлемые для решения моей проблемы. В качестве первого критерия красоты программного кода я определяю возможность его многократного использования без тени сомнения в его способности выдавать нужные результаты. Иными

словами, меня в первую очередь беспокоит не то, как он выглядит, а то, что я могу с ним сделать.

Это не означает, что я не ценю красоту элементов реализации кода; разумеется, я их тоже оцениваю по достоинству, и несколько позже в этой главе намерен рассмотреть критерии и примеры внутренней красоты кода. Моя позиция заключается в том, что если код отвечает моим в чем-то нетрадиционным понятиям красоты его полезных качеств, то вряд ли стоит обращать внимание на тонкости его реализации. На мой взгляд, такой код содействует решению одной из наиболее важных задач производства: возможности его совместного использования с другими разработчиками, не заставляя их проводить его анализ и разбираться в тонкостях его работы. Красивый код похож на красивую машину. Необходимость открывать капот и осматривать ее механизмы возникает у вас крайне редко. Вместо этого вы радуетесь ее внешнему виду и верите в то, что она довезет вас куда угодно.

Чтобы таким же образом радоваться программному коду, он должен быть сконструирован так, чтобы было абсолютно понятно, как его можно использовать, и совсем нетрудно сообразить, как его можно привязать к решению ваших собственных проблем, а также легко можно было бы проверить, правильно ли вы его используете.

Представление библиотеки лаборатории ЦЕРН

Я уверен, что разработанная в ЦЕРНе (европейском центре ядерных исследований) математическая библиотека может служить готовым примером красивого кода. Код, имеющийся в этой библиотеке, выполняет алгебраические операции, интегрирует функции, решает дифференциальные уравнения и помогает разобраться в большом количестве физических проблем. Он был написан более 30 лет назад и широко использовался все эти годы. Часть библиотеки ЦЕРНа, относящаяся к решению задач линейной алгебры, превратилась в библиотеку LAPACK, которая и содержит рассматриваемый здесь код. В настоящее время разработкой библиотеки LAPACK занимается ряд университетов и организаций.

Я пользовался этим кодом еще в молодости и до сих пор не перестаю им восхищаться. Его красота заключается в том, что он содержит множество сложных, прошедших очень тщательную проверку и довольно трудно воспроизведимых математических алгоритмов. Вы можете с полной уверенностью ими воспользоваться, абсолютно не заботясь о том, как именно они работают.

Высокая точность и надежность библиотеки является ответом на вопрос, почему даже по прошествии стольких лет она остается той математической библиотекой, которую выбирает каждый, кто нуждается в надежном решении уравнений с приемлемой точностью. Существуют, конечно, и другие общедоступные математические библиотеки, но по доказанной точности и надежности они не могут сравниться с библиотекой ЦЕРНа.

В первой части данной главы рассматривается внешняя красота этого программного кода: то, что придает ему эту точность и надежность, вызывая у разработчиков желание использовать его снова и снова, и те элементы, которые максимально облегчают его повторное использование. Во второй части анализируется внутренняя красота элементов его реализации.

Внешняя красота

Если вам когда-либо приходилось решать систему линейных уравнений или выполнять сопоставимые по сложности математические операции, то вы уже знаете, что очень часто код, написанный для выполнения этой задачи, не выдает правильных результатов. Одной из самых существенных проблем всех математических библиотек является то, что ошибки округлений и операций с плавающей запятой приводят к нестабильным и неверным результатам.

Если вы разрабатываете математическую библиотеку, вам нужно тщательно определить рабочий диапазон каждого алгоритма. Нужно так написать каждый алгоритм, чтобы он придерживался этих условий, и при этом еще найти способ компенсации ошибок округления. Это может стать весьма непростой задачей.

В библиотеке ЦЕРНа алгоритмы определены с исключительной точностью. В большинстве случаев при поиске какой-нибудь стандартной процедуры привлекает внимание описание ее предназначения. В принципе неважно, на каком языке написана процедура. На самом деле эти процедуры написаны на Фортране, но имеют интерфейс, позволяющий вызывать их практически из любого другого места. Это придает им дополнительную привлекательность. В некотором смысле стандартные процедуры можно рассматривать в качестве своеобразного черного ящика: вам совершенно безразлично, что творится у него внутри, интерес представляет лишь выдача приемлемых результатов на основе введенных вами данных. Довольно четко определено предназначение каждой процедуры, условие, при котором она сохраняет работоспособность, характеристики воспринимаемых ею данных и те ограничения, которые должны накладываться на входные данные с целью получения корректных результатов.

Взглянем, к примеру, на стандартную процедуру SGSV, принадлежащую библиотеке LAPACK и предназначенную для решения системы линейных уравнений для ленточной матрицы. Если вы пытаетесь получить числовое решение системы линейных уравнений, то используете различные алгоритмы. Для разных областей определения лучше работают разные алгоритмы, и для выбора лучшего из них требуется знать структуру матрицы. К примеру, вам нужно воспользоваться одним алгоритмом для решения задачи, если вы располагаете ленточной матрицей (у которой большинство элементов расположено близко к диагонали) и каким-нибудь другим алгоритмом, если у вас в распоряжении имеется разреженная матрица (большинство элементов которой равны нулю).

Поскольку различные подпрограммы оптимизированы под разные ситуации, то решение, какой из них лучше воспользоваться, по сути зависит от структуры имеющейся матрицы. Тем не менее, чтобы понять область изменений,

вы должны понимать порядок ввода данных в эти процедуры. Иногда данные вводятся в форме матрицы. А иногда, к примеру в случае с линейной матрицей, вы посыпаете их в виде очень узкого массива. Каждая из этих процедур и соответствующие требования по их использованию довольно ясно описаны в самой библиотеке:

```
SUBROUTINE SGGSV( N, KL, KU, NRHS, AB, LDAB, IPIV, B, LDB, INFO )
*
* -- LAPACK driver routine (version 2.0) --
* Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
* Courant Institute, Argonne National Lab, and Rice University
* March 31, 1993
*
* .. Скалярные параметры ..
INTEGER INFO, KL, KU, LDAB, LDB, N, NRHS
*
* .. Параметры, представленные массивами ..
INTEGER IPIV( * )
REAL AB( LDAB, * ), B( LDB, * )
*
*
* Назначение
* -----
*
* SGGSV выдает решение для системы линейных уравнений, представленной в
* вещественных числах  $A * X = B$ , где  $A$  – это линейная матрица  $N$ -ного
* порядка с  $KL$  поддиагоналями и  $KU$  главными диагоналями, а  $X$  и  $B$  – это
* матрицы размером  $N$  на  $NRHS$ .
*
* Декомпозиция LU с частичным выбором ведущего элемента и перестановкой
* строк используется, чтобы разложить на множители  $A$ , то есть  $A = L * U$ , где
*  $L$  – это результат перестановки и часть нижних треугольных матриц с  $KL$ 
* поддиагоналяй, и  $U$  – это верхний треугольник с  $KL+KU$  наддиагоналяй.
* Разложенная на множители форма  $A$  затем используется для вычисления системы
* линейных уравнений  $A * X = B$ .
*
* Параметры
* -----
*
* N   (ввод) INTEGER
* Количество линейных уравнений, то есть порядок матрицы  $A$ .  $N \geq 0$ .
*
* KL  (ввод) INTEGER
* Количество поддиагоналей в пределах полосы  $A$ .  $KL \geq 0$ .
*
* KU  (ввод) INTEGER
* Количество наддиагоналей в пределах полосы  $A$ .  $KU \geq 0$ .
*
* NRHS (ввод) INTEGER
* Число правых частей, то есть число столбцов распределенной
* подматрицы  $B$ .  $NRHS \geq 0$ .
*
* AB  (ввод /вывод) REAL массив, размерность (LDAB,N)
* На входе матрица  $A$  находится в ленточном хранилище, в строках от
*  $KL+1$  до  $2*KL+KU+1$ ; строки массива с 1 по  $KL$  устанавливать не нужно.
*  $j$ -тый столбец  $A$  хранится в  $j$ -том столбце массива  $AB$ , как изложено
* ниже:
```



```

* Проверка входных параметров.
*
INFO = 0
IF( N.LT.0 ) THEN
  INFO = -1
ELSE IF( KL.LT.0 ) THEN
  INFO = -2
ELSE IF( KU.LT.0 ) THEN
  INFO = -3
ELSE IF( NRHS.LT.0 ) THEN
  INFO = -4
ELSE IF( LDAB.LT.2*KL+KU+1 ) THEN
  INFO = -6
ELSE IF( LDB.LT.MAX( N, 1 ) ) THEN
  INFO = -9
END IF
IF( INFO.NE.0 ) THEN
  CALL XERBLA( 'SGBSV ', -INFO )
  RETURN
END IF
*
* Выполнение LU-разложения ленточной матрицы A.
*
CALL SGBTRF( N, N, KL, KU, AB, LDAB, IPIV, INFO )
IF( INFO.EQ.0 ) THEN
*
* Вычисление системы A*X = B, переписывая B на X.
*
CALL SGBTRS( 'No transpose', N, KL, KU, NRHS, AB, LDAB, IPIV,
$           B, LDB, INFO )
END IF
RETURN
*
* End of SGBSV
*
END

```

Первое, на что следует обратить внимание в коде подпрограммы SGBSV, — на пространный комментарий в самом начале, в котором описывается ее предназначение и использование. Фактически этот комментарий полностью повторяет страницу руководства, посвященную этой подпрограмме. Наличие в коде полной документации по использованию подпрограммы является весьма важным обстоятельством, поскольку это объединяет внутреннюю структуру подпрограммы с ее использованием. Во многих других случаях я заметил, что описание в руководстве и документация в коде не имеют ничего общего. Я думаю, что практика сочетаемости этих двух источников информации является одним из факторов красоты кода.

Следом за начальными комментариями идет описание подпрограммы, в котором изложены подробности используемого в ней алгоритма. Это помогает любому пользователю кода понять, что код будет делать и как он будет действовать. Затем следует подробное описание параметров с точным определением их диапазонов.

Параметр **AB** является одним из тех, которые представляют интерес для рассмотрения. Он содержит элементы матрицы **A**. Поскольку матрица является

линейной, в ней содержится множество нулевых значений, которые не сгруппированы поблизости от диагонали. В принципе, входные данные, используемые подпрограммой, могут быть представлены в виде двумерного массива, соответствующего размерности матрицы. Но это приведет к неоправданной трате памяти. Вместо этого параметр **AB** содержит только ненулевые элементы матрицы, которые находятся рядом с ее диагональю.

Формат параметра **AB** не только экономит память, но преследует также и другую цель. В этой подпрограмме алгоритм использует свойства системы уравнений для решения проблемы более эффективным способом. Это означает, что алгоритм полагается на пользователя, который в качестве входных данных должен обеспечить правильный тип матрицы. Если параметр **AB** содержит все элементы матрицы, то для одного или нескольких элементов за пределами линии могут быть случайно установлены ненулевые значения. Это приведет к ошибочному решению. Выбранный для **AB** формат делает совершение этой ошибки невозможным. Это было сделано преднамеренно и внесло свой вклад в красоту кода.

Параметр **AB** играет также и другую роль: он является не только входным, но и выходным параметром. Благодаря этой особенности конструкции решается другая проблема. Поскольку подпрограмма повторно использует пространство памяти, в котором размещается исходная программа, разработчики этого кода гарантируют, что подпрограмма будет работать до тех пор, пока у исходной программы будет достаточно памяти.

Если код был написан так, что для подпрограммы требовалась бы дополнительно выделенная память, то она могла бы и не запуститься, если система не могла бы выделить больший объем памяти. Это проблема приобретает особую актуальность при решении действительно большой системы уравнений, и подпрограмме для работы требуется значительное пространство памяти. Типовой код освобождает от этой проблемы, поскольку он был написан так, что подпрограмма способна возвратить результат до тех пор, пока исходная программа располагает достаточным пространством памяти для размещения задачи. Это очень красивое решение.

Перед тем как перейти к другим параметрам, я хочу продолжить рассмотрение этого вопроса. За свою жизнь я видел немало созданного программного кода. Зачастую разработчики пишут код и подсознательно устанавливают собственные для него ограничения. Чаще всего они ограничивают круг проблем, который он может решать. Это происходит в результате следующего мыслительного процесса:

1. мне нужно что-то написать;
2. я быстро все напишу и посмотрю, будет ли это работать;
3. как только все заработает, я подведу код к решению реальной проблемы.

Этот процесс подводит разработчиков к выстраиванию в коде ограничений, очень часто приводящих к неочевидным ошибкам, на выявление которых порой уходят годы. В ходе этого процесса разработчики обычно устанавливают явные или неявные ограничения на размеры той проблемы, которую они могут решить. Например, явные ограничения могут выразиться в определении большого

пространства данных, которое было бы достаточно большим для решения всех проблем. Это плохое, но достаточно легко выявляемое решение. Неявное ограничение может выразиться в неверном использовании динамической памяти, например, при написании кода, который вынуждает программу по представлению задачи проводить динамическое распределение памяти для ее решения. Для больших задач это может вызывать проблемы переполнения памяти и оказывать существенное влияние на производительность. Потери производительности основаны на зависимости программы от утилиты операционной системы, занимающейся страничной подкачкой. Если алгоритм потребует больших вычислительных ресурсов и ему понадобятся данные из множества различных участков памяти, то программа будет постоянно обращаться к подкачке и выполняться очень медленно.

Другой пример подобных проблем проявляется в программах, использующих систему управления базами данных. Если программа создана по вышеупомянутым канонам, она может при обработке данных постоянно обращаться к базе данных. Программист может думать, что программа выполняет простые и быстрые операции, а на самом деле они будут неэффективными из-за постоянных обращений к базе данных.

Ну и какой же урок можно извлечь из всего этого? При создании красивого кода нужно всегда думать о его масштабируемости. Вопреки существующим представлениям, масштабируемость не возникает за счет оптимизации кода; скорее всего, она возникает за счет использования правильного алгоритма. При анализе кода могут быть выявлены признаки низкой производительности, но основная причина проблем быстродействия может быть прослежена, как правило, при обращении к самой конструкции. Подпрограмма SGBSV была сконструирована так, чтобы не иметь таких проблем производительности, и это еще одна составляющая ее красоты.

Теперь, после просмотра других входных параметров, становится понятным, что в них используется тот же принцип, который применен к массиву АВ. Последний параметр — INFO — является механизмом предоставления информации об ошибках. Интересно наблюдать за тем, как эта диагностика доводится до пользователя. Вполне возможно, что система уравнений не имеет решения, и для этого здесь тоже есть сообщение. Заметьте, что в параметре INFO может быть сообщение не только об ошибке и об удачном выполнении, но и диагностическое сообщение, помогающее определить характер проблемы.

В программном коде, который пишется в настоящее время, этот вопрос зачастую упускается. В наши дни код часто создается с расчетом на обработку позитивных случаев использования: он пишется для выполнения действий, подробно описанных в спецификации.

Для типового кода это означает, что он будет работать в том случае, если система уравнений имеет решение. Но реальность более сурова. На практике если программа столкнется с системой уравнений, не имеющей решения, ее работа может прерваться, код выведет информацию о состоянии ядра или вызовет исключение. Это типичная ошибка в определении требований, касающихся неожиданных случаев применения.

Сегодня многие системы программируются в расчете на минимальный объем работы; после чего однажды во время работы они не смогут «подстроиться» под те обстоятельства, которые никто для них изначально не предусматривал. Похожая проблема возникает при неудачном определении технических требований, касающихся качественной обработки ошибок и других неожиданных ситуаций. Адекватный ответ на исключительные обстоятельства является важной составляющей надежности приложения и должен рассматриваться как основное функциональное требование.

При составлении спецификаций разработчикам следует понимать, что эти спецификации, как правило, остаются незавершенными. Разработчик должен обладать глубоким пониманием проблемы, чтобы иметь возможность расширить технические требования, учитывая случаи нетривиального использования и неожиданного развития событий, которые должны быть реализованы в целях разумной работы кода. Наша типовая подпрограмма служит одним из примеров того, что получается в том случае, когда проводится тщательный анализ. Эта подпрограмма будет либо работать по предназначению, либо сообщит вам о том, что она не в состоянии выполнить эту работу, но она никогда вас не подведет. В этом и заключается ее красота.

Теперь взглянем на раздел подпрограммы **Дополнительные подробности**. В этом разделе описывается порядок использования памяти и поясняется, что пространство при выполнении внутренних операций используется как рабочее. Это хороший пример красиво выполненного кода, который будет рассмотрен в следующем разделе **«Внутренняя красота»**.

Другой пример внешней красоты программного кода заключается в том, что ко многим подпрограммам библиотеки ЦЕРНа прилагаются простые тесты и примеры программ. Это весьма важная составляющая красоты. Вы должны быть в состоянии определить, сможет ли код справиться с тем, что он предположительно должен делать в вашем приложении. Разработчики этой библиотеки написали тестовые программы, как она вызывается для тех или иных данных. Вы можете воспользоваться этими тестовыми программами, чтобы проверить, будут ли корректные результаты получены для ваших данных, проникаясь, таким образом, доверием к библиотеке.

Красота этой конструкции в том, что тесты не только дают возможность поять, в каких условиях вы можете использовать подпрограммы, но также дают и пример проверки правильности, который позволяет вам приобрести уверенность и узнать, что происходит, без обращения к самому коду.

Внутренняя красота

Теперь рассмотрим подробности реализации кода.

Красота, заключенная в лаконичности и простоте

Я уверен, что красивый код должен быть коротким, и я считаю, что длинный, сложный код, как правило, уродлив. Подпрограмма SGBSV — один из лучших

примеров красоты короткого кода. Он начинается с быстрой проверки соответствия введенных параметров, затем осуществляются два вызова, которые логически придерживаются математического алгоритма. Что делается с помощью этого кода, становится понятным с первого взгляда: он начинается с выполнения LU-разложения на множители с помощью подпрограммы SGBTRF, затем выполняется решение системы с помощью подпрограммы SGBTRS.

Этот код очень легко читается. Чтобы понять, что делает этот код, совсем не обязательно детально изучать сотни его строк. Главная задача разбита на две подзадачи, которые помещены в подсистему. Заметьте, что подсистема придерживается тех же самых конструкторских установок по использованию памяти, что и основная система. Это очень важная и красивая особенность конструкции. Подпрограммы подсистемы повторно используются в различных «ведущих» подпрограммах (подпрограмму SGSV называют ведущей). Тем самым создается иерархическая система, поддерживающая повторное использование кода. Это тоже красиво. Многократное использование кода существенно сокращает усилия, необходимые для его разработки, тестирования и поддержки. Фактически это один из наилучших способов повышения продуктивности разработчиков и уменьшения их нервной нагрузки. Проблема в том, что многократное использование обычно дается нелегко.

Зачастую код настолько сложен и труден для чтения, что разработчики считают, что проще создать код заново, чем повторно использовать код, разработанный кем-то другим. Удачная конструкция и понятный, лаконичный код являются жизненно важным условием для повторного использования.

К сожалению, большая часть кода, созданного сегодня, не соответствует этим канонам. Основной объем созданного сегодня кода имеет структуру наследования, которая вселяет надежду на то, что она внесет в код некоторую ясность. Тем не менее я должен признаться, что потратил немало времени, уставившись на несколько строк подобного кода... и никак не мог понять, для чего он предназначен. В таком коде не видно красоты; это неудачный код с запутанной конструкцией. Если взглянув на подборку имен и несколько строк кода, вы не можете ничего сказать о его предназначении, значит? код слишком сложен.

Красивый код должен быть понятным. Я не люблю читать код, который был создан, чтобы похвастаться глубиной знания языка, и я не хочу разбираться в содержимом 25 файлов для того, чтобы понять, для чего предназначен конкретный фрагмент кода. Сопровождать код комментарием совсем не обязательно, но его предназначение должно быть явно выражено, и в любом его действии не должно быть неясных моментов. Проблема нового кода, созданного в наши дни, особенно на языке C++, состоит в том, что разработчики используют такое количество наследований и перегрузок, что становится практически невозможно сказать о том, для чего этот код на самом деле предназначен, почему он это делает и насколько это правильно. Чтобы все это определить, вам нужно разобраться во всей иерархии наследований и перегрузок. Если действие является разновидностью сложных перегрузок, то этот код я не могу считать красивым.

Красота, заключенная в экономии

Мой следующий критерий красивого кода касается степени продуманности работы кода на компьютере. Этим я пытаюсь сказать, что при разработке красивого кода всегда нужно брать в расчет, что он будет работать на компьютере, и у этого компьютера есть свои ограничения. Как ранее было замечено, у компьютера существуют скоростные ограничения, способности работать лучше с вещественными или целыми числами и у него имеется строго определенный объем оперативной памяти. Красивый код должен учитывать эти реально существующие ограничения. Зачастую создатель программного кода предполагает наличие безграничной памяти, неограниченной скорости компьютера и т. д. Этот код нельзя считать красивым; в нем кроются несбыточные надежды. В красивом коде выражается бережное отношение к таким вещам, как использование памяти и ее повторное использование там, где это возможно.

Рассмотрим, к примеру, подпрограмму LU-разложения SGBTRF, которая представляет собой второй уровень подпрограмм. Для экономии места я опустил начальные комментарии в заголовке и копировал все то, что не будет рассматриваться (полноценную подпрограмму можно посмотреть по адресу <http://www.netlib.org/lapack/explore-html/sgbtrf.f.html>):

```
SUBROUTINE SGBTRF( M, N, KL, KU, AB, LDAB, IPIV, INFO )
*
* -- LAPACK routine (version 2.0) --
* Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
* Courant Institute, Argonne National Lab, and Rice University
* February 29, 1992
```

Начальные комментарии. описание параметров

```
* Тестирование входных параметров.
*
INFO = 0
IF( M.LT.0 ) THEN
```

Проверка параметров

```
    CALL XERBLA( 'SGBTRF', -INFO )
    RETURN
END IF
*
* Быстрое возвращение, если возможно.
*
```

```

    IF( M.EQ.0 .OR. N.EQ.0 )
$ RETURN
*
* Определение размера блока для данной среды
*
* NB = ILAENV( 1, 'SGBTRF', ' ', M, N, KL, KU )
*
* Размер блока не должен превышать лимита, установленного размерами
* локальных массивов WORK13 и WORK31.
*
* NB = MIN( NB, NBMAX )
*
* IF( NB.LE.1 .OR. NB.GT.KL ) THEN
*
* Использование разблокированного кода
*
      CALL SGBTF2( M, N, KL, KU, AB, LDAB, IPIV, INFO )
ELSE
*
* Использование блокированного кода
*
* Обнуление наддиагональных элементов рабочего массива WORK13
*
DO 20 J = 1, NB
  DO 10 I = 1, J - 1
    WORK13( I, J ) = ZERO
10  CONTINUE
20 CONTINUE
*
* Обнуление поддиагональных элементов рабочего массива WORK31
*
DO 40 J = 1, NB
  DO 30 I = J + 1, NB
    WORK31( I, J ) = ZERO
30  CONTINUE
40 CONTINUE
*
* Гауссово исключение с частичным выбором ведущего элемента
*
* Установка заполненных элементов в столбцах с KU+2 по KV в нуль
*
DO 60 J = KU + 2, MIN( KV, N )
  DO 50 I = KV - J + 2, KL
    AB( I, J ) = ZERO
50  CONTINUE
60 CONTINUE
*
* JU - индекс последнего столбца, затронутого текущей стадией
* разложения на множители
*
JU = 1
*
DO 180 J = 1, MIN( M, N ), NB
  JB = MIN( NB, MIN( M, N )-J+1 )
*
* Активная часть матрицы разделена

```

```

*
*   A11 A12 A13
*   A21 A22 A23
*   A31 A32 A33
*
*   Здесь A11, A21 и A31 обозначают текущий блок из JB столбцов.
*   который будет разложен на множители. Количество строк в декомпозиции
*   равно, соответственно, JB, I2, I3, а количество столбцов равно
*   JB, J2, J3. Наддиагональные элементы из A13
*   и поддиагональные элементы из A31 выходят за рамки полосы.
*
I2 = MIN( KL-JB, M-J-JB+1 )
I3 = MIN( JB, M-J-KL+1 )
*
*   J2 и J3 вычисляются после обновления JU.
*
*   Разложение на множители текущего блока из JB столбцов
*
DO 80 JJ = J, J + JB - 1
*
*       Установка заполненных элементов в столбце JJ+KV в нуль
*
IF( JJ+KV.LE.N ) THEN
    DO 70 I = 1, KL
        AB( I, JJ+KV ) = ZERO
70    CONTINUE
END IF
*
*   Поиск ведущего элемента и проверка на сингулярность. KM – это
*   количество поддиагональных элементов в текущем столбце.
*
KM = MIN( KL, M-JJ )
JP = ISAMAX( KM-1, AB( KV+1, JJ ), 1 )
IPIV( JJ ) = JP + JJ - J
IF( AB( KV+JP, JJ ).NE.ZERO ) THEN
    JU = MAX( JU, MIN( JJ+KU+JP-1, N ) )
    IF( JP.NE.1 ) THEN
*
*           Применение перестановки для столбцов с J thru J+JB-1
*
        IF( JP+JJ-1.LT.J+KL ) THEN
*
            CALL SSWAP( JB, AB( KV+1+JJ-J, J ), LDAB-1,
$               AB( KV+JP+JJ-J, J ), LDAB-1 )
        ELSE
*
*           Перестановка затрагивает столбцы с J по JJ-1
*           матрицы A31, хранящейся в рабочем массиве WORK31
*
            CALL SSWAP( JJ-J, AB( KV+1+JJ-J, J ), LDAB-1,
$               WORK31( JP+JJ-J-KL, 1 ), LDWORK )
            CALL SSWAP( J+JB-JJ, AB( KV+1, JJ ), LDAB-1,
$               AB( KV+JP, JJ ), LDAB-1 )
        END IF
    END IF
$    END IF

```

```

*      Вычисление сомножителей
*
CALL SSCAL( KM, ONE / AB( KV+1, JJ ), AB( KV+2, JJ ) .
$           1 )
*
```

Продолжение решения

```

170      CONTINUE
180      CONTINUE
END IF
*
RETURN
*
* Завершение SGBTRF
*
END

```

Подпрограмма опять начинается с проверки параметров, после чего переходит к решению задачи, которое сопровождается проверкой оптимизации, которая рассматривает размер задачи, чтобы определить, может ли она быть решена в «кэшируемых» массивах WORK13 и WORK31, или ее нужно отослать на нижний уровень, для более сложных операций. Это превосходный пример кода, созданного с учетом реальности, для компьютера, имеющего определенные ограничения. Рабочий массив должен быть приспособлен под стандартную память компьютера, на котором решается задача; при решении задач достаточно небольшого размера, это может препятствовать снижению производительности от возможной постраничной подкачки. Задачи, превосходящие этот размер, настолько велики, что без потерь производительности просто не обойтись.

Красота, заключающаяся в последовательности

Решение предыдущей проблемы предусматривало пошаговое изложение алгоритма. Чтение этого кода очень похоже на чтение книги, поскольку в нем можно легко разобраться. Части задачи, свойственные другим алгоритмам, используются повторно, а части, которые могли бы усложнить код, помещены в подпрограммы. В результате получается вполне простая и понятная последовательность.

Каждый шаг этой последовательности соотносится с математическим выражением. На каждом шаге в коде описывается, что ожидается от системы низшего уровня, и вызывается эта система. Основная подпрограмма, являющаяся ведущей, разветвляется на подпрограммы низшего уровня, каждая из которых разветвляется на подпрограммы еще более низшего уровня и т. д. Эта последовательность отображена на рис. 15.1.

Это превосходный пример применения принципа «разделяй и властвуй» к проектированию кода. При каждом перемещении на низшую ступень преодоле-

левается менее значительная проблема, в результате чего можно сфокусироваться на вполне определенных обстоятельствах, которые делают код более лаконичным и целенаправленным. Если проблема заключается в компьютерной памяти, то алгоритм немедленно приступит к ее решению, как было рассмотрено ранее. Если этой проблемы не существует, то он перейдет на следующий уровень подпрограмм, и т. д.

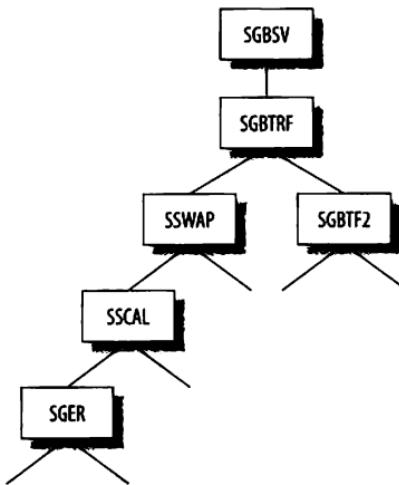


Рис. 15.1. Логическое разделение задач на подпрограммы

В результате подпрограмма, требующая весьма интенсивных вычислений, может быть написана на ассемблере, а затем оптимизирована под архитектуру. Другое преимущество такой конструкции состоит в том, что над кодом может одновременно трудиться множество людей, поскольку каждая подпрограмма является независимой, а потому вполне определенной.

Заключение

В итоге я уверен, что красивый код должен быть кратким, абсолютно определенным, экономным и созданным с учетом существующей действительности. Тем не менее я считаю, что подлинной проверкой на красоту — для кода, так же как и для искусства — является проверка временем. За эти годы написана масса программного кода, но лишь незначительная его часть используется и по сей день, несмотря на то что уже прошли годы со времени его написания. То, что код библиотеки ЦЕРНа используется до сих пор, спустя более 30 лет со времени его создания, подтверждает его истинную красоту.

16 Модель драйверов ядра Linux: преимущества совместной работы

Грег Кроах-Хартман (Greg Kroah-Hartman)

В модели ядра Linux предпринята попытка создания общесистемного древа из всех различных типов устройств, управляемых операционной системой. Основная структура использованных для этого данных и кода с годами изменялась от очень простой системы, предназначавшейся для управления несколькими устройствами, до весьма масштабируемой системы, способной управлять всевозможными различными типами устройств, с которыми нужно взаимодействовать в реальном мире.

С годами ядро Linux развивалось, справляясь со все большим количеством различных типов устройств¹, основы ядра нужно было изменять и совершенствовать, чтобы добиться более простых и управляемых способов оперирования существующим диапазоном устройств.

Почти все устройства состоят из двух различных частей: физической части, определяющей способ общения операционной системы с устройством (будь то шина PCI, SCSI, ISA, USB и т. д.), и виртуальной части, определяющей, как операционная система представляет устройство пользователю, чтобы оно могло быть соответствующим образом задействовано (клавиатура, мышь, видео, звук и т. д.).

Вплоть до ядра версии 2.4 каждая физическая часть устройств управлялась своей частью кода, зависящей от типа шины. Этот код шины отвечал за широкий диапазон различных задач, и код для каждой отдельной шины не взаимодействовал с каким-нибудь другим кодом шины.

В 2001 году Пэт Мочел (Pat Mochel) работал над решением проблемы управления электропитанием в ядре Linux. Он пришел к выводу, что для правильного включения или выключения электропитания отдельных устройств ядро должно знать о связи различных устройств друг с другом. Например, привод USB-диска должен быть отключен от питания до того, как будет отключена карта PCI-контроллера для отключаемого USB-контроллера, чтобы данные

¹Сейчас Linux поддерживает больше различных устройств и процессоров, чем какая-либо другая операционная система, имевшаяся за всю историю вычислений.

этого устройства были правильно сохранены. Для решения этой задачи ядро должно знать о древе всех присутствующих в системе устройств, показывающем, какое устройство находится под управлением другого устройства и порядок, в котором все они были подключены.

Примерно в то же самое время я натолкнулся на другую проблему, связанную с устройствами: Linux не всегда правильноправлялся с устройствами. Я хотел, чтобы два моих USB-принтера всегда имели одни и те же имена, независимо от того, который из них был включен первым или в какой очередности они были обнаружены ядром Linux.

Для некоторых других операционных систем этот вопрос был решен за счет размещения в ядре небольшой базы данных для обработки имен устройств или в них была предпринята попытка экспорттировать всевозможные уникальные характеристики устройства посредством разновидности файловой системы `devfs`¹, которая могла быть использована для непосредственного обращения к устройству. Для Linux размещение базы данных внутри ядра было неприемлемым решением. Кроме того, реализация файловой системы `devfs` для Linux содержала ряд хорошо известных и неустранимых врожденных условий, мешающих почти всем распространяемым пакетам Linux рассчитывать на ее использование. К тому же решение, связанное с `devfs`, вынуждало пользователя применять определенную методику именования. Хотя некоторые считали это преимуществом, эта методика шла вразрез с опубликованными стандартами образования имен в Linux и никому не позволяла использовать при желании другую методику образования имен.

Мы с Пэтром поняли, что обе наши проблемы могут быть решены за счет унифицированного драйвера и модели устройства внутри ядра Linux. Такая унифицированная модель не была какой-то новой идеей, поскольку в других операционных системах подобная модель когда-то уже была реализована. Теперь настал черед Linux последовать их примеру. Такая модель позволяла бы создавать древо всех устройств, а также позволяла бы программам в пользовательском пространстве, находящемся за пределами ядра, оперировать постоянным обозначением любого устройства, любым желаемым для пользователя способом.

В этой главе будет рассмотрено развитие структуры данных и поддержка функций внутри ядра Linux, выполняющих эту работу, и то, как это развитие привело к изменениям, которые никто не мог ожидать в самом начале процесса разработки.

Скромное начало

Для начала в качестве «базового» класса для всех устройств в ядре была создана простая структура — `struct device`. Вначале эта структура имела следующий вид:

¹ Применение `devfs` — один из способов, использующихся в операционной системе для демонстрации пользователям всевозможных доступных ему устройств. Это делается за счет отображения всего разнообразия имен устройств, а иногда и ограниченных отношений между этими устройствами.

```

struct device {
    struct list_head node;          /* узел в дочернем списке */
    struct list_head children;
    struct device *parent;
    char   name[DEVICE_NAME_SIZE];  /* описательная ascii-строка */
    char   bus_id[BUS_ID_SIZE];     /* позиция на родительскойшине */

    spinlock_t      lock;           /* блокировка устройства,
                                      обеспечивающая невозможность
                                      одновременного доступа к нему с
                                      различных уровней. */
    atomic_t refcount;             /* refcount обеспечивает
                                      существование устройства
                                      необходимое количество времени */

    struct driver_dir_entry * dir;

    struct device_driver *driver;   /* какой драйвер распределил это
                                      устройство */
    void      *driver_data;        /* данные, принадлежащие драйверу */
    void      *platform_data;      /* Данные, характерные для платформы
                                      (например ACPI, BIOS данные,
                                      существенные для устройства) */

    u32 current_state;            /* Текущее функциональное состояние
                                      Если говорить об ACPI, это D0-D3,
                                      D0 в режиме полного
                                      функционирования, и D3 в
                                      отключенном состоянии. */

    unsigned char *saved_state;    /* сохраненное состояние
                                      устройства */
};

};

```

При каждом создании этой структуры и ее регистрации в драйверном основании ядра создается новая запись в виртуальной файловой системе, показывающая устройство и любые имеющиеся в нем отличительные свойства. Это дает возможность всем установленным в системе устройствам отображаться в пространстве, доступном пользователю в том порядке, в котором они были подключены. Эта виртуальная файловая система теперь называется sysfs и может быть просмотрена на Linux-машине в каталоге /sys/devices. Пример этой структуры, отображающей несколько PCI- и USB-устройств, выглядит следующим образом:

```

$ tree -d /sys/devices/
/sys/devices/pci0000:00/
|-- 0000:00:00.0
|-- 0000:00:02.0
|-- 0000:00:07.0
|-- 0000:00:1b.0
|   |-- card0
|   |   |-- adsp
|   |   |-- audio
|   |   |-- controlC0
|   |   |-- dsp

```

```

|   |-- mixer
|   |-- pcmC0D0c
|   |-- pcmC0D0p
|   |-- pcmC0D1p
|   -- subsystem -> ../../../../../../class/sound
-- driver -> ../../../../../../bus/pci/drivers/HDA_Intel
-- 0000:00:1c.0
|   |-- 0000:00:1c.0:pcie00
|   |-- 0000:00:1c.0:pcie02
|   |-- 0000:00:1c.0:pcie03
|   |-- 0000:01:00.0
|       '-- driver -> ../../../../../../bus/pci/drivers/sky2
|   -- driver -> ../../../../../../bus/pci/drivers/pcieport-driver
-- 0000:00:1d.0
|   '-- driver -> ../../../../../../bus/pci/drivers/uhci_hcd
-- usb2
|   |-- 2-0:1.0
|       '-- driver -> ../../../../../../bus/usb/drivers/hub
|       '-- subsystem -> ../../../../../../bus/usb
|           '-- usbdev2.1_ep81
|       '-- driver -> ../../../../../../bus/usb/drivers/usb
|           '-- usbdev2.1_ep00
-- 0000:00:1d.2
|   '-- driver -> ../../../../../../bus/pci/drivers/uhci_hcd
-- usb4
|   |-- 4-0:1.0
|       '-- driver -> ../../../../../../bus/usb/drivers/hub
|           '-- usbdev4.1_ep81
|   |-- 4-1
|       '-- 4-1:1.0
|           '-- driver -> ../../../../../../bus/usb/drivers/usbhid
|               '-- usbdev4.2_ep81
|           '-- driver -> ../../../../../../bus/usb/drivers/usb
|               '-- power
|               '-- usbdev4.2_ep00
|   |-- 4-2
|       '-- 4-2.1
|           '-- 4-2.1:1.0
|               '-- driver -> ../../../../../../bus/usb/drivers/usbhid
|                   '-- usbdev4.4_ep81
|           '-- 4-2.1:1.1
|               '-- driver -> ../../../../../../bus/usb/drivers/usbhid
|                   '-- usbdev4.4_ep82
|               '-- driver -> ../../../../../../bus/usb/drivers/usb
|                   '-- usbdev4.4_ep00
|       '-- 4-2.2
|           '-- 4-2.2:1.0
|               '-- driver -> ../../../../../../bus/usb/drivers/usb1p
|                   '-- usbdev4.5_ep01
|                   '-- usbdev4.5_ep81
|               '-- driver -> ../../../../../../bus/usb/drivers/usb
|                   '-- usbdev4.5_ep00
|       '-- 4-2:1.0
|           '-- driver -> ../../../../../../bus/usb/drivers/hub
|               '-- usbdev4.3_ep81
|       '-- driver -> ../../../../../../bus/usb/drivers/usb

```

```

|   |   '-- usbdev4.3_ep00
|   '-- driver -> ../../bus/usb/drivers/usb
|   '-- usbdev4.1_ep00
...

```

Чтобы воспользоваться этой структурой, нужно, чтобы вы встроили ее в другую структуру, заставляя новую структуру в некотором смысле стать «наследником» базовой структуры:

```

struct usb_interface {
    struct usb_interface_descriptor *altsetting;

    int act_altsetting:           /* активация дополнительной настройки */
    int num_altsetting:          /* количество дополнительных параметров
                                  настройки */
    int max_altsetting:          /* общее количество распределенной
                                  памяти */
    struct usb_driver *driver:    /* драйвер */
    struct device dev:           /* информация об особенностях интерфейса
                                  устройства */
};


```

Драйвер ядра функционирует путем раздачи указателей на struct device, оперируя основными полями, находящимися в этой структуре и распространяемыми, таким образом, на все типы устройств. Когда указатель передан различным функциям в коде, определенном для соответствующей шины, он должен быть преобразован к реальному типу содержащейся в нем структуры. Для обработки этого преобразования код, разработанный под определенную шину, направляет указатель обратно, на исходную структуру, взяв за основу ее место в памяти. Это делается с помощью следующей функциональной макрокоманды:

```
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) * __mptr = (ptr); \
    (type *)((char *)__mptr - offsetof(type, member));})


```

Для тех, кто не может разобраться в арифметике ее указателей, представленной на языке Си, эта макрокоманда требует объяснений. К примеру, предыдущий struct usb_interface мог преобразовать указатель к элементу структуры struct device, вернувшись к исходному указателю с помощью следующего кода:

```

int probe(struct device *d) {
    struct usb_interface *intf;

    intf = container_of(d, struct usb_interface, dev);
...
}


```

где d — это указатель на struct device.

Расширение только что показанной макрокоманды container_of приводит к созданию следующего кода:

```

intf = ({
    const typeof( ((struct usb_interface *)0)->dev ) * __mptr = d;
    (struct usb_interface *)((char *)__mptr - offsetof(struct \
usb_interface, dev));
});


```

Чтобы разобраться в этом коде, нужно помнить, что `dev` — это элемент структуры `struct usb_interface`. В первой строке макрокоманды устанавливается указатель, который указывает на переданный коду `struct device`. Во второй строке макрокоманды происходит поиск реального размещения в памяти `struct usb_interface`, к которому мы хотим получить доступ.

Итак, имея известный тип структуры `dev`, макрокоманда может быть сокращена до следующего вида:

```
intf = ({  
    const struct device *_mptr = d;  
    (struct usb_interface *)( (char *)_mptr - offsetof(struct \  
        usb_interface, dev));  
});
```

На основании недавно показанного определения `struct usb_interface` переменная `dev`, по всей вероятности, поместила 16 байт в структуру на 32-битном процессоре. Компилятор автоматически это пересчитает с помощью макрокоманды `offsetof`. Теперь помещение этой информации в макрокоманду приведет к следующему результату:

```
intf = ({  
    const struct device *_mptr = d;  
    (struct usb_interface *)( (char *)_mptr - 16));  
});
```

Макрокоманда `container_of` теперь сокращена до простого арифметического указателя, вычитающего 16 из исходного указателя, чтобы получить желаемый указатель `struct usb_interface`. Компилятор во время своей работы со всем этим быстро справляется.

Благодаря этому очень простому методу ядро Linux позволяет обычным Си-структурям получить очень мощный способ приобретения наследуемости и управляемости. Но его мощность может быть реализована только в том случае, если вы точно знаете, что именно делаете.

Если вы заметили, что в процессе выполнения не проводится проверка типов, обеспечивающая, что указатель, изначально переданный как `struct device`, действительно относился к типу `struct usb_interface`. Обычно большинство систем, занимающихся подобной манипуляцией с указателями, имеют также поле в базовой структуре, в котором определяется тип обрабатываемого указателя, чтобы можно было отлавливать ошибки программирования, возникшие по недосмотру. Это также позволяет создавать код с динамически определяемым типом указателя и производить на основании типа различные действия.

Разработчики ядра Linux приняли решение не проводить подобных проверок или определений типа. Эти проверки типов могут отлавливать основные ошибки программирования на начальной стадии разработки, но позволяют программистам допускать оплошности, которые в дальнейшем могут привести к куда более коварным проблемам, разобраться с которыми будет не так-то легко.

Отсутствие проверки в процессе выполнения заставляет разработчиков, занимающихся этими указателями, быть абсолютно уверенными, что они знают, какой тип указателя обрабатывается и посыпается по системе. Несомненно, в какие-то моменты разработчик действительно хочет, чтобы был какой-нибудь

способ определения, на какой тип struct device он смотрит, но когда задача достаточно отлажена, это желание в конечном счете проходит.

Может ли такое отсутствие проверки типа быть достаточно хорошим, чтобы все это можно было назвать «красивым кодом»? После пятилетнего срока работы с этой системой, я думаю, что можно. Это обстоятельство не позволяет мелким просчетам распространяться по ядру и заставлять всех быть предельно точными в своих логических заключениях, никогда не возвращаясь к каким-либо проверкам типа, которые могли бы предупредить последующее возникновение ошибок.

Здесь я должен отметить, что над этими частями ядра работало относительно небольшое количество разработчиков — те, кто занимался кодированием подсистем для общепринятых шин, — и они ожидали наработать значительный положительный опыт; именно поэтому в данном случае и не было никакой поддержки в форме проверки типов.

Благодаря этому методу наследования основной структуры struct device в процессе разработки ядра версии 2.5 были унифицированы всевозможные драйверные подсистемы. Теперь они совместно используют общий код ядра, позволяющий показать пользователю порядок взаимосвязи всех устройств. Это позволило создать такие инструментальные средства, как udev, которое решало вопрос постоянства наименования устройств в небольшой программе в пространстве пользователя, и инструменты управления электропитанием, способные проходить древо устройств и отключать их в правильной последовательности.

Превращение в еще более мелкие части

После переделки исходного ядра драйвера другой разработчик ядра системы, Ал Виро (Al Viro), работал над устранением ряда проблем, касающихся подсчета ссылок на объект на уровне виртуальной файловой системы.

Основная проблема, связанная со структурами в многопоточных программах, написанных на языке Си, заключалась в значительных трудностях с определением условий безопасного освобождения памяти, используемой той или иной структурой. Ядро Linux — это программа с широким применением многопоточности, которая должна справляться и с крайне активными пользователями, и с большими количествами одновременно запущенных процессоров. В связи с этим подсчет ссылок на любую структуру должен быть найден более чем одним потоком.

Структура struct device была одной из таких структур с подсчетом ссылок. Она имела отдельное поле, которое использовалось для определения условий безопасности ее освобождения:

```
atomic_t refcount;
```

При инициализации структуры это поле устанавливалось в 1. Когда же любой код намеревался воспользоваться структурой, он должен был сначала увеличить показания счетчика, вызвав функцию get_device, которая проверяет

правильность счетчика ссылок и увеличивает значение счетчика ссылок на структуру:

```
static inline void get_device(struct device * dev)
{
    BUG_ON(!atomic_read(&dev->refcount));
    atomic_inc(&dev->refcount);
}
```

Следуя этой же логике, когда поток заканчивает работу со структурой, он уменьшает значение счетчика ссылок, вызывая несколько более сложную функцию put_device:

```
void put_device(struct device * dev)
{
    if (!atomic_dec_and_lock(&dev->refcount, &device_lock))
        return;

    /* Указание драйверу на то, что он должен закончить работу.
     * Заметьте, что устройство, скорее всего, нами не задействовано,
     * поэтому у драйвера есть удобный случай его отключить ...
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

Функция уменьшает на единицу счетчик ссылок, а затем, если это был последний пользователь объекта, указывает объекту на необходимость закончить его работу и вызвать функцию, которая предварительно была настроена на отключение его от системы.

Ал Виро (Al Viro) понравилась унификация структуры struct device, виртуальная файловая система, показывающая все множество различных устройств и то, как они связаны друг с другом, а также автоматический подсчет ссылок. Единственная проблема была в том, что ядро его виртуальной файловой системы не работало на «устройства» и не имело «драйверов», которые могли бы привязываться к этим объектам. Поэтому он решил немного по-новому взглянуть на вещи и упростить код.

Ал уговорил Пэта создать нечто под названием struct kobject. Эта структура обладала основными свойствами структуры struct device, но была меньше и не имела связи с «драйвером» и «устройством». Она содержала следующие поля:

```
struct kobject {
    char name[KOBJ_NAME_LEN];
    atomic_t refcount;
    struct list_head entry;
    struct kobject *parent;
    struct subsystem *subsys;
    struct dentry *dentry;
};
```

Эта структура имела тип пустого объекта. Она обладала лишь самой общей способностью подпадать под подсчет ссылок на нее и быть вставленной в иерархию объектов. Теперь структура struct device могла включать эту маленькую «базовую» структуру struct kobject, чтобы унаследовать всю ее функциональность:

```

struct device {
    struct list_head g_list; /* node in depth-first order list */
    struct list_head node; /* node in sibling list */
    struct list_head bus_list; /* node in bus's list */
    struct list_head driver_list;
    struct list_head children;
    struct list_head intf_list;
    struct device *parent;

    struct kobject kobj;
    char bus_id[BUS_ID_SIZE]; /* position on parent bus */
    ...
}

```

Макрокоманда `container_of` используется для возврата назад от глубинного `kobject` к главному `struct device`:

```
#define to_dev(obj) container_of(obj, struct device, kobj)
```

Пока шел этот процесс разработки, многие другие специалисты увеличили запас прочности ядра Linux, позволив ему постепенно брать под свое управление все большее и большее количество процессоров, запускаемых в едином системном образе¹. По этой причине многие другие разработчики добавляли счетчики ссылок к своим структурам, чтобы они могли как следует управлять пространством задействованной ими памяти. Каждый разработчик должен был дублировать возможность инициализации, увеличения, уменьшения значения счетчика и завершения работы со структурой. Поэтому было решено, что эти элементарные функции должны быть убраны из `struct kobject` и преобразованы в свою собственную структуру. Так родилась структура `struct kref`:

```

struct kref {
    atomic_t refcount;
}:
```

`struct kref` обладает лишь тремя простыми функциями: `kref_init` для инициализации счетчика ссылок, `kref_get` для увеличения значения счетчика и `kref_put` для уменьшения этого значения.

Первые две функции слишком просты, а вот последняя несколько интереснее:

```

int kref_put(struct kref *kref, void (*release)(struct kref *kref))
{
    WARN_ON(release == NULL);
    WARN_ON(release == (void (*) (struct kref *))kfree);

    if (atomic_dec_and_test(&kref->refcount)) {
        release(kref);
        return 1;
    }
    return 0;
}
```

У функции `kref_put` есть два параметра: указатель на `struct kref`, чей счетчик ссылок нужно уменьшить, и указатель на функцию, которую вы хотите вызвать, если это была последняя ссылка, числящаяся за объектом.

¹ Текущий рекорд по количеству процессоров, работающих под управлением Linux в едином системном образе, равен 4096, поэтому работу по масштабируемости системы можно признать успешной.

Первые две строки функции были добавлены сразу же после того, как struct kref была добавлена к ядру, поскольку некоторые программисты пытались обойти счетчик ссылок, либо не передавая функции освобождения пространства памяти (`release`) указателя, либо, когда они поняли, что ядро на это пожалуется, не передавая указателя основной функции `kfree`. (Как ни грустно, но в наши дни стало недостаточно даже этих двух проверок. Некоторые программисты в своих попытках полностью проигнорировать счетчик ссылок взялись за создание пустых функций освобождения пространства, которые вообще ничего не делали. К сожалению, в Си отсутствует простой способ определения, делает ли что-нибудь указатель на функцию внутри самой этой функции, иначе и эта проверка тоже была бы добавлена к `kref_put`.)

После проведения этих двух проверок счетчик ссылок автоматически уменьшается на единицу, и если это была последняя ссылка, вызывается функция освобождения пространства памяти и возвращается 1. Если это была не последняя ссылка на объект, возвращается 0. Это возвращаемое значение используется только для того, чтобы определить, была ли вызывающая процедура последней из тех, кто задействовал объект, не находится ли объект все еще в памяти (она не может гарантировать, что объект все еще существует, поскольку кто-то еще мог прийти и избавиться от него уже после того, как вызов выдал возвращаемое значение).

С созданием struct kref структура struct kobject претерпела изменения, чтобы ею можно было воспользоваться:

```
struct kobject {
    char          name[KOBJ_NAME_LEN];
    struct kref   kref;
};


```

Встраивания всех этих различных структур в другие структуры привело к тому, что рассмотренная ранее исходная struct `usb_interface` теперь содержала struct `device`, которая содержала struct `kobject`, которая содержала struct `kref`. А кто говорил, что на языке Си трудно реализовать объектно-ориентированное программирование...

Расширение масштаба до тысяч устройств

Поскольку Linux работает на всем, от сотовых телефонов, радиоуправляемых вертолетов, настольных компьютеров и серверов и до 73 % самых больших в мире суперкомпьютеров, масштабирование модели драйверов играло весьма важную роль и всегда присутствовало в нашем сознании. По мере развития разработки нас радовало то, что основные структуры, задействованные в поддержке устройств, struct `kobject` и struct `devices`, были относительно невелики. Количество устройств, подключенных к большинству систем, было прямо пропорционально размеру системы. Поэтому небольшая, встроенная система имела всего несколько — от одного до десяти — различных устройств в подключенном и зафиксированном в ее древе устройств состояния. Более крупные

«промышленные» системы имели намного больше подключенных устройств, но у этих систем также имелось в избытке значительное количество памяти, поэтому увеличивающееся количество устройств все еще задействовало очень небольшую часть памяти по отношению к общему объему, задействованному ядром системы.

К сожалению, эта удобная модель масштабирования оказалась полностью несостоятельной, когда дело дошло до одного из классов «промышленных» систем, универсальной вычислительной машины s390. Этот компьютер мог запускать Linux в виртуальном разделе (одновременно до 1024 экземпляров на одной машине) и обладал огромным количеством подключенных к нему различных устройств для хранения информации. В целом система располагала большим объемом памяти, но каждый виртуальный раздел мог иметь только небольшую часть от этого объема. Каждому виртуальному разделу нужно было видеть весь арсенал устройств хранения (вполне обычное количество могло составлять 20 000), в то время как ему распределялось лишь несколько сотен мегабайт оперативной памяти.

Довольно быстро обнаружилось, что на этих системах древо устройств поглощало огромный процент памяти, который никогда не возвращался пользовательским процессам. Настало время посадить драйверную модель на диету, и работать над этой проблемой пришли некоторые очень умные разработчики ядра IBM.

То, что они обнаружили, сначала вызывало удивление. Оказалось, что главная структура `struct device` занимала всего лишь около 160 байт (на системе с 32-битным процессором). При 20000 устройствах, подключенных к системе, задействованная оперативная память разрасталась всего лишь до 3–4 Мб, и спрятаться с таким объемом не составляло никакой проблемы. А самым большим пожирателем объема памяти была основанная на RAM ранее упомянутая файловая система `sysfs`, с помощью которой все эти устройства отображались в пользовательском пространстве. Для каждого устройства `sysfs` создавала и `struct inode`, и `struct dentry`. Обе эти структуры были довольно тяжеловесными, `struct inode` задействовала 256 байт, а `struct dentry` – около 140 байт¹.

Для каждой `struct device` создавалась по крайней мере одна `struct dentry` и одна `struct inode`. В общем получалось так, что создавалось множество отдельных копий этой файловой структуры, по одной на каждый виртуальный файл для каждого устройства, подключенного к системе. К примеру, отдельное блочное устройство могло создавать более 10 различных виртуальных файлов, а значит, одиночная структура в 160 байт могла в конечном счете занимать 4 Кб. В системе с 20 000 устройств на виртуальную файловую систему понапрасну тратилось около 80 Мб. Эта память потреблялась ядром и не могла уже быть использована какими-нибудь пользовательскими программами, даже если им никогда не нужно было просматривать информацию, хранящуюся в `sysfs`.

Решение этой проблемы заключалось в переделке кода `sysfs`, чтобы поместить эти структуры, `struct inode` и `struct dentry`, в кэш-память ядер, создавать их

¹ С тех пор обе эти структуры подверглись сокращению, и в современных версиях ядра они занимают меньший объем памяти.

на лету, при обращениях к файловой системе. По сути решение состояло в динамическом создании каталогов и файлов на лету, по мере того как пользователь путешествовал по древу, вместо того чтобы заранее размещать всю информацию при первоначальном создании устройства. Поскольку эти структуры находятся в основной кэш-памяти ядра, если программы в пользовательском пространстве создают в системе трудности с памятью или с другими частями ядра, кэш-память освобождается, и память возвращается тому процессу, который на данный момент в ней нуждается. Все это было сделано за счет переделки внутреннего кода `sysfs` и не касалось главных структур `struct device`.

Свободно присоединяемые мелкие объекты

Модель драйверов Linux наглядно продемонстрировала, как за счет создания множества мелких объектов, каждый из которых предназначен для хорошего выполнения всего одной задачи, язык Си может быть использован для создания модели кода, обладающей глубокой объектной ориентацией.

Эти объекты могут встраиваться в другие объекты, без использования в процессе выполнения какой-либо идентификации типов, создавая очень мощное и гибкое дерево объектов. И основываясь на реальном использовании этих объектов, можно сказать, что они задействуют минимальный объем памяти, позволяя ядру Linux быть достаточно гибким, чтобы с одной и той же кодовой базой можно было успешно работать в диапазоне от очень маленьких встроенных систем и до самых больших в мире суперкомпьютеров.

Разработка этой модели также показывает два очень интересных и мощных аспекта того направления, в котором идет разработка ядра Linux.

Во-первых, процесс имеет строго выраженный последовательный характер. Как только изменяются требования к ядру и система, на которой оно работает, тоже претерпевает изменения, разработчики находят способы извлечения различных частей модели, чтобы при необходимости сделать их более эффективными. Это отвечает основным эволюционным потребностям системы по выживанию в этих средах.

Во-вторых, история управления устройствами показывает, что этот процесс происходит в очень тесном сотрудничестве. Разные коллективы разработчиков независимо друг от друга генерируют идеи для улучшения и расширения различных аспектов ядра. Имея в своем распоряжении исходный код, другие могут оценить цели разработчиков в точном соответствии с их формулировкой, а затем помочь изменить код в том направлении, которое затеявшими разработку никогда не рассматривалось. В конечном итоге, за счет поиска общих решений, которые не могут быть видны кому-то одному, получается продукт, отвечающий целям многих разработчиков, работающих в различных направлениях.

Эти две особенности разработки помогли системе Linux превратиться в одну из самых гибких и мощных из всех когда-либо созданных операционных систем. И они гарантируют, что до тех пор пока разработка будет продолжаться в том же стиле, Linux сохранит свои позиции.

17 Иной уровень косвенного подхода

Диомидис Спинеллис (Diomidis Spinellis)

Знаменитое высказывание, приписываемое Батлеру Лампсону (Butler Lampson), ученому, предвидевшему еще в 1972 году появление современного персонального компьютера, гласит: «Все проблемы в компьютерной науке могут быть решены на ином уровне косвенного подхода». Отголосок этого высказывания возникал в моем сознании в различных ситуациях: когда меня вынуждали беседовать с секретарем, вместо того чтобы общаться с нужным мне человеком, когда я сначала ехал на восток во Франкфурт, чтобы потом улететь на запад, в Шанхай или Бангалор, и – конечно же – когда я изучал исходный код сложной системы.

Наше путешествие мы начнем с рассмотрения проблемы типичной операционной системы, которая поддерживает различные форматы файловой системы. Операционная система может использовать данные, постоянно хранящиеся в своей родной файловой системе, на компакт-диске или на флэш-карте USB. Эти устройства хранения могут, в свою очередь, воспользоваться различными организациями файловой системы: NTFS или ext3fs, которые работают на Windows или Linux, ISO-9660, которая используется для компакт-дисков, и традиционной FAT-32, часто используемой для флэш-карты USB. Каждая файловая система использует различные структуры данных для управления свободным пространством, для хранения метаданных файла и для организации файлов в каталоги. Поэтому каждая система требует различных кодов для каждой файловой операции (открытия – open, чтения – read, записи – write, поиска – seek, закрытия – close, удаления – delete и т. д.).

От кода к указателям

Я вступал в зрелый возраст во времена, когда разные компьютеры чаще всего имели несовместимые файловые системы, вынуждая меня переносить данные с одной машины на другую, связывая их по последовательным портам. Поэтому

меня неизменно поражала возможность читать на моем ПК флэш-карту, записанную на камере. Давайте рассмотрим, как операционная система структурировала бы код для обращения к различным файловым системам. Один из подходов мог бы заключаться в использовании для каждой операции оператора `switch`. Рассмотрим пример гипотетического системного вызова чтения — `read` под управлением операционной системы FreeBSD. Его интерфейс на стороне ядра мог бы выглядеть следующим образом:

```
int VOP_READ(
    struct vnode *vp,      /* Файл, из которого происходит чтение */
    struct uio *uio,       /* Определение буфера */
    int ioflag,            /* Флаги, определяемые для ввода-вывода (I/O) */
    struct ucred *cred)   /* Учетные данные пользователя */

{
    /* Гипотетическая реализация */
    switch (vp->filesystem) {
        case FS_NTFS:      /* Код, специфичный для NTFS */
        case FS_ISO9660:   /* Код, специфичный для ISO-9660 */
        case FS_FAT32:     /* Код, специфичный для FAT-32 */
        /* [...] */
    }
}
```

Этот подход связал бы вместе код для различных файловых систем, ограничивая применение принципа модульности. Хуже того, добавление поддержки новой файловой системы потребовало бы модификации кода реализации каждого системного вызова и перекомпиляции ядра. Кроме этого, добавление шага обработки ко всем операциям файловой системы (к примеру, проверки соответствия учетных данных удаленного пользователя) потребовало бы также незастрахованной от ошибок модификации каждой операции с одним и тем же стереотипом кода.

Как вы могли уже догадаться, решаемая нами задача нуждается в каких-то дополнительных уровнях косвенного подхода. Рассмотрим, как решает эти проблемы операционная система FreeBSD, которой я восхищаюсь за зрелость разработки основного кода. Каждая файловая система определяет поддерживаемые ею операции в виде функций, а затем инициализирует структуру `vop_vector` с указателями на эти функции. Вот как выглядят некоторые поля структуры `vop_vector`:

```
struct vop_vector { struct vop_vector *vop_default;
    int (*vop_open)(struct vop_open_args *);
    int (*vop_access)(struct vop_access_args *);
```

а вот как файловая система ISO-9660 инициализирует структуру:

```
struct vop_vector cd9660_vnodeops = {
    .vop_default = &default_vnodeops,
    .vop_open = cd9660_open,
    .vop_access = cd9660_access,
    .vop_bmap = cd9660_bmap,
    .vop_cachedlookup = cd9660_lookup,
    .vop_getattr = cd9660_getattr,
    .vop_inactive = cd9660_inactive,
```

```

.vop_ioctl = cd9660_ioctl,
.vop_lookup = vfs_cache_lookup,
.vop_pathconf = cd9660_pathconf,
.vop_read = cd9660_read,
.vop_readdir = cd9660_readdir,
.vop_readlink = cd9660_readlink,
.vop_reclaim = cd9660_reclaim,
.vop_setattr = cd9660 setattr,
.vop_strategy = cd9660_strategy,
};

}:

```

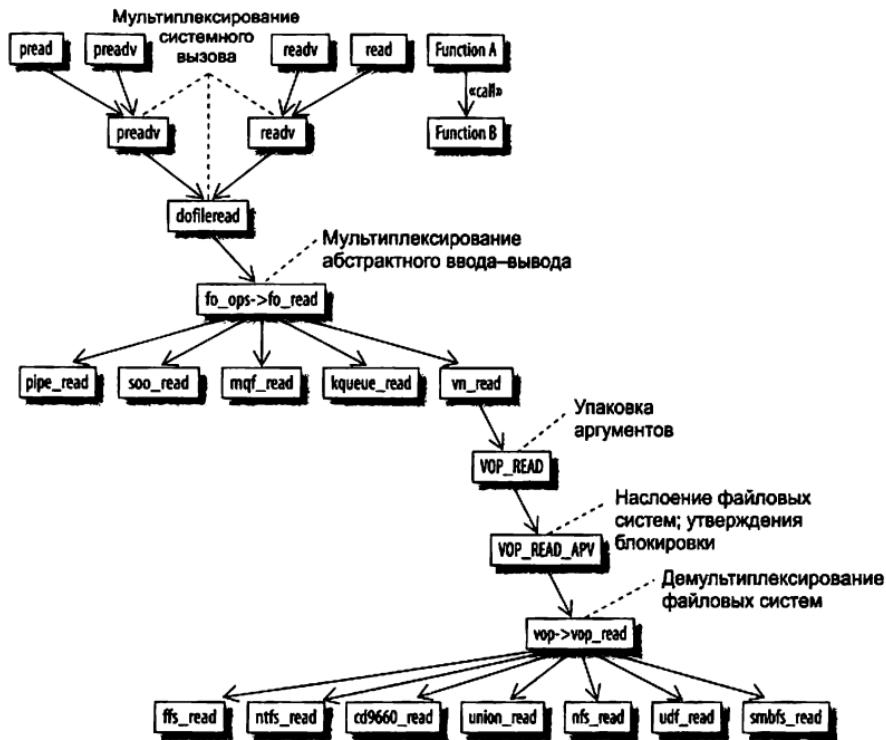


Рис. 17.1. Уровни косвенного подхода в реализации системного вызова чтения в FreeBSD

(Синтаксис .поле = значение — весьма привлекательное свойство стандарта C99, позволяющее полям структуры инициализироваться в произвольном порядке и хорошо читаться.) Учтите, что в предыдущем коде определены только 16 полей, в то время как полная структура vop_vector содержит 52 поля. К примеру, поле vop_write осталось неопределенным (получая значение NULL), поскольку запись в файлы файловой системой компакт-дисков ISO-9660 не поддерживается. Инициализация по одной такой структуре для каждого типа файловой системы (см. нижнюю часть рис. 17.1) облегчает последующую привязку этой структуры

к административным данным, связанным с каждым описателем файла. Затем в ядре FreeBSD реализация независимой от файловой системы части системного вызова чтения — `read` может принять следующий вид (см. рис. 17.1):

```
struct vop_vector *vop;
rc = vop->vop_read(a);
```

Итак, при чтении с компакт-диска, содержащего файловую систему ISO-9660, предыдущий вызов через указатель фактически привел бы к вызову функции `cd9660_read`:

```
rc = cd9660_read(a);
```

От аргументов функций к аргументам указателей

Большинство Unix-подобных операционных систем, среди которых FreeBSD, Linux и Solaris, для отделения реализации файловой системы от кода, который обращается к ее содержимому, используют указатели функций. Примечательно, что FreeBSD также использует косвенный подход для чтения аргументов функций.

Когда я впервые столкнулся с вызовом функции `vop->vop_read(a)`, показанной в предыдущем разделе, то задался вопросом, что из себя представлял аргумент `a` и что случилось с оригинальными четырьмя аргументами гипотетической реализации увиденной ранее функции `VOP_READ`. Разобравшись, я понял, что ядро использует другой уровень косвенного подхода, чтобы наслонить файловые системы на верхушки друг друга до произвольной глубины. Это наслонение позволяет файловой системе предложить ряд служб (среди которых прозрачные просмотры, сжатие и шифрование) на основе служб другой, низлежащей файловой системы. Для поддержки этого свойства два механизма осуществляют вполне разумную совместную работу: один разрешает отдельной обходной функции модифицировать аргументы любой функции `vop_vector`, в то время как другой позволяет всем неопределенным функциям `vop_vector` быть направленными на низлежащий уровень файловой системы.

Оба механизма в действии показаны на рис. 17.2. На нем изображены три файловые системы, наложенные на верхушки друг друга. На самом верху находится файловая система `umapfs`, установки для которой определяет системный администратор, чтобы распределить полномочия пользователей. Это может пригодиться, если система, в которой создан конкретный диск, использует другие пользовательские идентификаторы (ID). Например, администратору может потребоваться, чтобы пользователь с ID 1013 появился в низлежащей файловой системе как пользователь с ID 5325.

Под верхней файловой системой расположена Berkeley Fast Filesystem (`ffs`), файловая система, способная эффективно обращаться с ресурсами памяти и времени и используемая по умолчанию при обычной установке FreeBSD. В свою очередь, при выполнении большинства своих операций `ffs` зависит от кода реализации оригинальной 4.2 BSD файловой системы `ufs`.

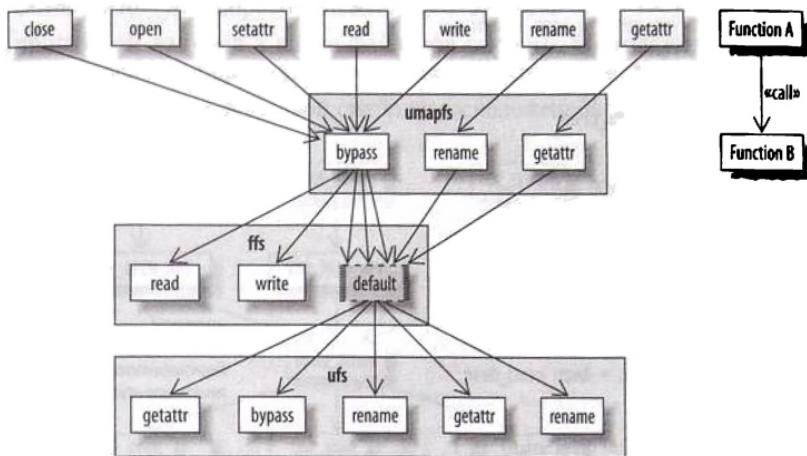


Рис. 17.2. Пример наслойения файловой системы

В примере, показанном на рисунке, большинство системных вызовов проходят через общую функцию обхода (`bypass`) в `umapfs`, которая распределяет пользовательские полномочия. Лишь некоторые системные вызовы, среди которых переименование — `rename` и получение атрибутов — `getattr`, имеют в `umapfs` свою собственную реализацию. Уровень `ffs` обеспечивает оптимизированную реализацию системных вызовов чтения и записи — `read` и `write`; оба из которых зависят от уровня файловой системы, работающего более эффективно, чем тот, который используется `ufs`. Большинство других операций, открытие — `open`, закрытие — `close`, получение атрибутов — `getattr`, установка атрибутов — `setattr` и переименование — `rename`, обрабатываются обычным способом. Таким образом, запись `vop_default` в структуре `ffs_vop_vector` направляет все эти функции на вызов низлежащей `ufs`-реализации. Например, системный вызов `read` будет пропущен через `umapfs_bypass` и `ffs_read`, тогда как вызов `rename` будет пропущен через `umapfs_rename` и `ufs_rename`.

Оба механизма, обхода и умолчания, пакуют четыре аргумента в единую структуру, чтобы обеспечить унификацию между различными функциями файловой системы, а также поддержать основу работы обходной функции. Этот красивый конструкторский шаблон очень легко проглядеть в хитросплетениях Си-кода, необходимого для его реализации.

Четыре аргумента пакуются в единую структуру, которая в качестве своего первого поля (`a_gen.a_desc`) содержит описатель содержимого структуры (`vop_read_desc`, в следующем примере кода). На рис. 17.1 видно, что системный вызов чтения файла — `read` в ядре FreeBSD инициирует вызов `vop_read`, который устанавливает соответствующие низкоуровневые аргументы и вызывает `VOP_READ`. Он запакует аргументы и вызовет `VOP_READ_APV`, который, в конце концов, вызывает `vop->vop_read` и вместе с ним реальную функцию чтения файловой системы:

```

struct vop_read_args {
    struct vop_generic_args a_gen;
    struct vnode *a_vp;
    struct uio *a_uio;
    int a_ioflag;
    struct ucred *a_cred;
};

static __inline int VOP_READ(
    struct vnode *vp,
    struct uio *uio,
    int ioflag,
    struct ucred *cred)
{
    struct vop_read_args a;

    a.a_gen.a_desc = &vop_read_desc;
    a.a_vp = vp;
    a.a_uio = uio;
    a.a_ioflag = ioflag;
    a.a_cred = cred;
    return (VOP_READ_APV(vp->v_op, &a));
}

```

Точно такой же непростой танец исполняется для вызова других функций `vop_vector` (`stat`, `write`, `open`, `close` и т. д.). Структура `vop_vector` также содержит указатель на функцию `bypass`. Эта функция получает запакованные аргументы, и после возможных модификаций некоторых из них (может быть таких, как отображение полномочий пользователя из одного административного домена в другом) передает управление соответствующей низлежащей функции для вызова, определенного через поле `a_desc`. Возьмем фрагмент, показывающий, как файловая система `nullfs` реализует функцию `bypass`. Она только дублирует часть существующей файловой системы в другое место глобального пространства имен файловой системы. Поэтому для большинства ее операций она может просто позволить своей функции `bypass` вызвать соответствующую функцию в низлежащей файловой системе:

```

#define VCALL(c) ((c)->a_desc->vdesc_call(c))
int
null_bypass(struct vop_generic_args *ap)
{
    /* ... */
    error = VCALL(ap);
}

```

В предыдущем коде макрокоманда `VCALL(ap)` подтолкнет `vnode`-операцию (например `VOP_READ_APV`), которая вызовет функцию `null_bypass`, находящуюся одним уровнем файловой системы ниже. Вы можете увидеть этот прием в действии на рис. 17.3.

Кроме того, `vop_vector` содержит поле под названием `default`, которое является указателем на структуру `vop_vector` низлежащего уровня файловой системы. Благодаря этому полю, если файловая система не осуществляет какого-то элемента функциональной нагрузки, запрос пропускается на более низкий уровень. Заполнением полей `bypass` и `default` своей структуры `vop_vector` файловая система может выбирать между:

- о самостоятельной обработкой входящего запроса;
- о пропуском запроса в адрес файловой системы, расположенной уровнем ниже, после модификации некоторых аргументов;
- о прямым вызовом файловой системы, расположенной уровнем ниже.

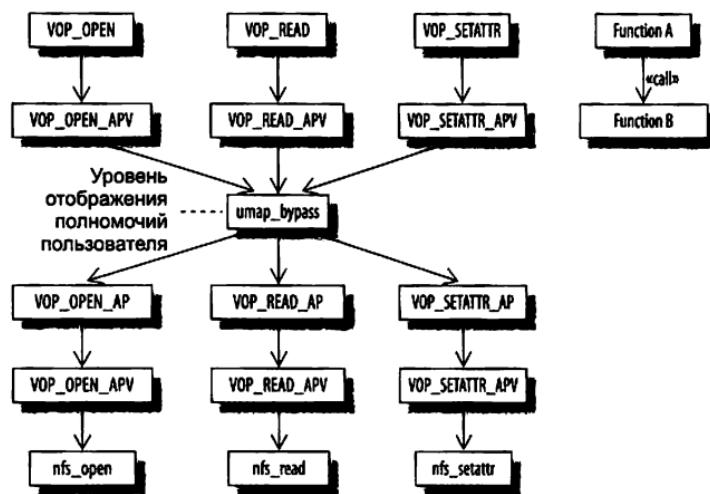


Рис. 17.3. Маршрутизация системных вызовов через функцию bypass

В своем воображении я это представляю в виде шарика, опускающегося вниз по наклонной плоскости и соприкасающегося с толкателями и вращателями сложного аппарата для игры в пинбол. Следующий пример реализации системы чтения показывает, как система определяет местонахождение вызываемой функции:

```

int
VOP_READ_APV(struct vop_vector *vop, struct vop_read_args *a)
{
    [...]
    /*
     * Проникновение на нижние уровни файловой системы для поиска
     * исполнителя востребованной функции или пропуска запроса дальше
     */
    while (vop != NULL &&
           vop->vop_read == NULL && vop->vop_bypass == NULL)
        vop = vop->vop_default;
    /* Вызов функции или пропуск */
    if (vop->vop_read != NULL)
        rc = vop->vop_read(a);
    else
        rc = vop->vop_bypass(&a->a_gen);
  
```

Изысканность кода состоит еще и в том, что на самом нижнем уровне находится файловая система, которая возвращает Unix-ошибку EOPNOTSUPP — «operation

not supported» (операция не поддерживается) для любой функции, которая не была реализована файловыми системами, являющимися его наследованиями. Это выходное отверстие нашего пинбола:

```
#define VOP_EOPNOTSUPP ((void*)(uintptr_t)vop_eopnotsupp)

struct vop_vector default_vnodeops = {
    .vop_default = NULL,
    .vop_bypass = VOP_EOPNOTSUPP,
};

int
vop_eopnotsupp(struct vop_generic_args *ap)
{
    return (EOPNOTSUPP);
}
```

От файловых систем к уровням файловой системы

В качестве конкретного примера разбиения файловой системы на уровни рассмотрим случай установки на компьютер удаленной файловой системы с использованием NFS-протокола (Network File System – сетевая файловая система). К сожалению, в нашем случае идентификаторы пользователей и группы на удаленной системе не соответствуют тем, которые используются на вашем компьютере. Тем не менее, поместив файловую систему `umapfs` поверх действующей NFS-реализации, мы можем определить через внешние файлы правильное распределение полномочий пользователя и группы. Рисунок 17.3 иллюстрирует, как некоторые системные вызовы ядер операционных систем, прежде чем продолжить свой путь к соответствующим функциям NFS-клиента, сначала направляются через обходную функцию `umapfs` – `umap_bypass`.

В отличие от функции `null_bypass`, реализация функции `umap_bypass` перед осуществлением вызова низлежащего слоя совершает некоторые действия. Структура `vop_generic_args`, передаваемая в качестве ее аргументов, содержит описание фактических параметров для каждой `vnode`-операции:

```
/*
 * Универсальная структура.
 * Может быть использована процедурами обхода для идентификации
 * универсальных параметров.
 */
struct vop_generic_args {
    struct vnodeop_desc *a_desc;
    /* по-видимому, далее следуют другие произвольные данные */
};

/*
 * Эта структура описывает выполняемую vnode-операцию.
 */
struct vnodeop_desc {
    char *vdesc_name; /* читаемое имя для отладки */
    int vdesc_flags; /* VDESC_* флаги */
```

```

vop_bypass_t *vdesc_call; /* Вызываемая функция */
*/
* Эти параметры используются обходной процедурой для
* отображения и определения местоположения аргументов.
* Creds и procs в процедуре обхода не нужны,
* но иногда они полезны. К примеру, для транспортных
* уровней.
* Польза Nameidata в том, что там содержатся полномочия
* пользователя.
*/
int *vdesc_vp_offsets; /* список, заканчивающийся
                         *DESC_NO_OFFSET */
int vdesc_vpp_offset; /* возвращение размещения vpp */
int vdesc_cred_offset; /* если есть, размещение cred */
int vdesc_thread_offset; /* если есть, размещение thread */
int vdesc_componentname_offset; /* если есть */
};

}:

```

Например, структура vnodeop_desc для аргументов, передаваемых операции vop_read, выглядит следующим образом:

```

struct vnodeop_desc vop_read_desc = {
    "vop_read",
    0,
    (vop_bypass_t *)VOP_READ_AP,
    vop_read_vp_offsets,
    VDESC_NO_OFFSET,
    VOPARG_OFFSETOF(struct vop_read_args,a_cred),
    VDESC_NO_OFFSET,
    VDESC_NO_OFFSET,
};

}:

```

Важно, что внутри аргументов вызова функции чтения отдельно от имени функции (используемого для отладочных целей) и вызываемой функции нижнего уровня (VOP_READ_AP) структура в своем поле vdesc_cred_offset содержит местонахождение поля данных с полномочиями пользователя (a_cred). Используя это поле, umap_bypass может отобразить полномочия любой vnode-операции с помощью следующего кода:

```

if (descp->vdesc_cred_offset != VDESC_NO_OFFSET) {
    credpp = VOPARG_OFFSETTO(struct ucred**,
        descp->vdesc_cred_offset, ap);
    /* сохранение старых значений */
    savecredpp = (*credpp);
    if (savecredpp != NOCRED)
        (*credpp) = crdup(savecredpp);
    credp = *credpp;
    /* Отображение всех идентификаторов в структуре полномочий. */
    umap_mapids(vp1->v_mount, credp);
}

```

Здесь мы имеем дело с данными, являющимися описанием формата других данных: в понятиях абстракции данных — с перенаправлением. Эти *метаданные* позволяют коду отображения полномочий манипулировать аргументами произвольных системных вызовов.

От кода к предметно-ориентированному языку

Вы могли заметить, что часть кода, связанного с реализацией системного вызова чтения, занимающаяся, в частности, упаковкой его аргументов в структуру или логикой вызова соответствующей функции, чрезвычайно стилизована и, по всей вероятности, повторена в подобных формах для всех 52 других интерфейсов. Другая деталь реализации, которую мы пока не рассматривали и которая может не дать мне спать по ночам, касается блокировки.

Операционные системы должны гарантировать, что различные процессы, работающие одновременно, не будут наступать друг другу на пятки при модификации данных в отсутствие координации своих действий. В современных многопоточных, многоядерных процессорах обеспечение непротиворечивости данных путем удержания одной взаимно исключающей блокировки для всех критических структур операционной системы (как это было в реализациях старых операционных систем) приведет к неприемлемому снижению производительности.

Поэтому в настоящее время блокировка устанавливается только на мелко-модульные объекты, такие пользовательские полномочия или отдельный буфер. Более того, поскольку установка и снятие блокировок могут быть весьма дорогостоящими операциями, в идеале, как только блокировка установлена, она не должна быть снята, если в ней тот час же опять возникнет необходимость. Эти спецификации блокировок лучше всего могут быть описаны через предусловия (какое состояние блокировки должно быть до входа в функцию), и постусловия (какое состояние блокировки должно быть по выходе из функции).

Можете представить, каким невероятно сложным делом может стать программирование и проверка правильности кода под таким давлением. К счастью для меня, для внесения некоторой ясности в эту картину может быть использован другой уровень косвенного подхода. Этот косвенный подход справляется как с избыточностью кода упаковки, так и с тонкостями блокировочной спецификации.

Изученные нами в ядре FreeBSD интерфейсные функции и структуры данных, среди которых VOP_READ_AP, VOP_READ_APV и vop_read_desc, не написаны просто на Си. Вместо этого для определения типов аргументов каждого вызова и их блокирующих пред- и постусловий использован предметно-ориентированный язык. Такой стиль реализации всегда вызывал у меня учащенное сердцебиение, поскольку мог приносить гигантский взлет производительности. Вот как выглядит фрагмент из спецификации системного вызова read:

```
#  
#% read vp L L L  
#  
vop_read {  
    IN struct vnode *vp;  
    IINOUT struct uio *uiio;  
    IN int ioflag;
```

```
IN struct ucred *cred;
};
```

На основе подобной спецификации awk-скрипт создает:

- Си-код для упаковки аргументов функций в единую структуру;
- объявления для структур, хранящих упакованные аргументы и функции, выполняющие работу;
- инициализированные данные, определяющие содержимое упакованных структур аргументов;
- шаблон Си-кода, который мы решили использовать для реализации уровней файловой системы;
- утверждения, для того чтобы проверить состояние блокировок при входах и выходах из функции.

В FreeBSD версии 6.1 реализация интерфейса вызова vnode занимает всего 588 строк предметно-ориентированного языка, которые расширяются в 4339 строк кода и объявлений на языке Си.

В компьютерном мире компиляция из специализированного высокоуровневого предметно-ориентированного языка в Си-код – довольно распространенное явление. К примеру, входные данные для генератора лексического анализа lex представляют собой файл, который отображает регулярные выражения в действиях; входные данные для парсер-генератора yacc представляют собой грамматику языка и вытекающие из нее правила. Обе системы (и их потомки flex и bison) генерируют Си-код, реализующий высокоуровневые спецификации. Более экстремальный случай касается ранней реализации языка программирования C++. Она состояла из препроцессора cfront, который должен был компилировать код C++ в Си-код.

Во всех этих случаях язык Си выступает в роли доступного языка ассемблирования. При должном использовании предметно-ориентированные языки увеличивают выразительность кода, а вместе с ней и продуктивность работы программиста. С другой стороны, используемый без особой надобности, невнятный предметно-ориентированный язык может только усложнить понимание, отладку и обслуживание системы.

Обработка утверждений, используемых в блокировке, требует дополнительных объяснений. Для каждого аргумента код составляет список состояний его блокировки для трех ситуаций: для входа в функцию, для успешного выхода из нее и для выхода из функции с ошибкой, создавая вполне понятное разделение задач. Например, предыдущая спецификация вызова read указывает на то, что аргумент up должен быть заблокирован во всех трех случаях. Возможны также и более сложные сценарии. Следующий фрагмент кода указывает на то, что аргументы fdvp и fvp вызова rename должны быть всегда разблокированы, но аргумент tdvp имеет блокировку, исключающую его обработку при вызове процедуры. Все аргументы должны быть разблокированы, когда функция завершает свою работу:

```
#  
#% rename fdvp U U U  
#% rename fvp U U U
```

```
#  
# rename tdvp E U U
```

Спецификация блокировки используется для оснащения Си-кода утверждениями при входе в функцию, при нормальном выходе из нее и при выходе с ошибкой. Например, код для выхода в функцию `rename` содержит следующие утверждения:

```
ASSERT_VOP_UNLOCKED(a->a_fdvp, "VOP_RENAME");
ASSERT_VOP_UNLOCKED(a->a_fvp, "VOP_RENAME");
ASSERT_VOP_ELOCKED(a->a_tdvp, "VOP_RENAME");
```

Хотя утверждения, подобные этим, не гарантируют безошибочности кода, но они, по крайней мере, обеспечивают индикацию раннего сбоя, которая поможет выявить ошибку во время тестирования системы, перед тем как она дестабилизирует систему и введет ее в режим, затрудняющий отладку. Чтение сложного кода, не имеющего утверждений, подобно наблюдению за воздушными акробатами, работающими без страховочной сетки: впечатляющее действие с небольшой помаркой может привести к весьма плачевному результату.

Мультиплексирование и демультиплексирование

Если вернуться к рис. 17.1, то можно увидеть, что обработка системного вызова чтения не начинается с `VOP_READ`. На самом деле эта функция вызывается из `vn_read`, который в свою очередь вызывается через указатель функции.

Этот уровень косвенного подхода имеет еще и другое назначение. Операционная система Unix и ее производные обрабатывают все источники ввода и вывода универсальным способом. Таким образом, вместо отдельных системных вызовов чтения из, скажем, файла, сокета или канала, системный вызов `read` может осуществлять чтение из любой из этих абстракций ввода–вывода. Я считаю такую конструкцию полезной и изящной; я часто на нее полагался, используя инструментарий таким способом, о котором его создатели, возможно, даже и не догадывались. (Это утверждение больше говорит о возрасте используемого мной инструментария, чем о моих творческих способностях.)

Косвенный подход, прослеживающийся в средней части рис. 17.1, отображает механизм FreeBSD, используемый для обеспечения независимости его высокуюровневого абстракционного ввода–вывода. Указатель функции, связанный с каждым описателем файла, ведет к коду, который будет обслуживать конкретный запрос: `pipe_read` – для каналов, `soo_read` – для сокетов, `mqf_read` – для очереди сообщения POSIX, `kqueue_read` – для очереди сообщений ядра и, наконец, `vn_read` – для текущих файлов.

До сих пор в нашем примере мы обнаружили два случая, где указатели функции используются для направления запроса к различным функциям. Обычно в таких случаях указатель функции используется для демультиплексирования (распределения) одиночного запроса по нескольким потенциальным исполнителям. Такое использование косвенного подхода настолько распространено,

что благодаря ему образуется важный элемент объективно-ориентированного языка в форме динамической диспетчеризации к различным методам подкласса. Для меня самостоятельная реализация динамической диспетчеризации в процедурном языке наподобие Си является отличительным признаком программиста высокого класса. (Другим таким же признаком является способность создавать структурированные программы на ассемблере или Фортране.)

Косвенный подход также часто представляется как способ разложения общих функциональных возможностей на составные элементы. Посмотрите на верхнюю часть рис. 17.1. Современные Unix-системы имеют четыре варианта унифицированного системного вызова read. Варианты системного вызова, начинающиеся с *p* (*pread*, *preadv*), разрешают вместе с вызовом иметь спецификацию позиции файла. Варианты, оканчивающиеся на *v* (*readv*, *preadv*), разрешают специфицировать вместо одного целый вектор запросов ввода-вывода. Хотя я считаю такое профилирование системы не слишком изящным и противоречивым духу Unix, похоже, что прикладные программисты впали от него в зависимость, в стремлении выжать каждый бит производительности из создаваемых ими веб-серверов или серверов баз данных.

Все эти вызовы используют совместно некий общий код. Чтобы избежать дублирования кода, в реализации FreeBSD вводится косвенный подход через дополнительные функции. Функция *kern_pread* обрабатывает общие части позиционированных вариантов системного вызова, а функция *kern_ready* обрабатывает остальные два системных вызова. Функциональность, общая для всех четырех вызовов, осуществляется другой функцией, *dofileread*. Я вполне могу представить себе ту радость, которую испытывают разработчики от того, что «выносят за скобки» общий для таких функций код за счет внедрения дополнительных уровней косвенности. У меня всегда поднимается настроение, если после осуществления перестройки составляющих элементов добавленных строк бывает меньше, чем удаленных.

Путь от вызова функции чтения в нашей пользовательской программе до движения головки жесткого диска для извлечения с пластины наших данных, слишком долг и извилист. В нашем описании мы не обсуждали все происходящее выше уровня ядра (виртуальные машины, буферирование, представление данных), или что происходит, когда файловая система обрабатывает запрос (опять же буферизация, драйверы устройств, представление данных). Интересно, что между этими двумя не охваченными нами концами есть забавная симметрия: они оба задействуют аппаратные интерфейсы (виртуальные машины, вроде JVM, наверху и реальные интерфейсы внизу), буферизацию (для минимизации системных вызовов наверху, и для оптимизации производительности оборудования внизу), и представление данных (для взаимодействия с местной спецификой пользователя наверху и для соответствия запросам физического уровня внизу). Создается впечатление, что косвенный подход используется практически везде куда только не кинешь взгляд. В показательном фрагменте, который мы рассматривали было девять уровней вызовов функций, два косвенных подхода, осуществляемых посредством указателей функций, и предметно-ориентированный язык, который дал нам вполне достаточное представление о своей мощности.

Уровни навсегда?

Примеры кода можно рассматривать до бесконечности, поэтому разумнее закончить наше обсуждение, отметив, что Лампсон (Lampson) приписывал высказывание, с которого началось наше исследование (все проблемы в компьютерной науке могут быть решены на ином уровне косвенного подхода), Дэвиду Уиллеру (David Wheeler), изобретателю подпрограмм. Примечательно, что Уиллер завершил свое высказывание другой фразой: «Но зачатую это ведет к возникновению другой проблемы». Несомненно, косвенность и разложение на уровни добавляют накладные расходы пространства и времени и могут усложнить понимание кода.

Накладные расходы времени и пространства чаще всего оказываются несущественными и вряд ли могут вызвать наше беспокойство. В большинстве случаев задержки, вызванные поиском еще одного указателя или вызовом подпрограммы, весьма незначительны на фоне более значительных обстоятельств. На поверку оказывается, что в настоящее время в современных языках программирования имеется тенденция постоянного проведения некоторых операций через косвенный уровень, чтобы придать им дополнительную гибкость. Так, к примеру, в Java и C# почти все обращения к объектам проходят через косвенный указатель, позволяющий проводить автоматическую уборку «мусора». Также в Java почти все вызовы методов экземпляра проходят диспетчеризацию через поисковую таблицу, в целях предоставления возможности наследованию классов проводить подмену метода в процессе выполнения программы.

Несмотря на накладные расходы, которыми обременены все обращения к объектам и вызовы методов, обе платформы чувствуют себя на рынке просто прекрасно, за что вам большое спасибо. В других случаях, при проведении оптимизации компиляторы удаляют ту косвенность, которая была заложена в кодами, разработчиками. Так, большинство компиляторов определяют, когда вызов функции обходится дороже, чем замена его действующим кодом, и выполняют это встраивание в автоматическом режиме.

С другой стороны, когда мы работаем на пороге производительности, косвенность может стать обременительной.

Одним из трюков заключается в том, что разработчики в попытках полностью воспользоваться возможностями интерфейса гигабитной сети вынуждены для ускорения работы его кода объединять функциональные возможности разных уровней сетевого стека, свертывая некоторые уровни абстракции. Но это можно отнести к экстремальным случаям.

С другой стороны, влияние, оказываемое косвенным подходом на понимание кода, вызывает весьма серьезный интерес, поскольку за последние 50 лет, в отличие от головокружительного взлета скорости работы центральных процессоров, возможности человека в понимании кода не претерпели существенных изменений. Поэтому сторонники гибких процессов советуют особо постараться, вводя использование уровней для того, чтобы справиться с некоторыми неопределенными, неконкретизированными требованиями, которые, в отличие от сегодняшних конкретных потребностей, могли бы, по нашим представлениям, неожиданно появиться в будущем. Как язвительно заметил Барт Смалдерс (Bart Smaalders), обсуждая шаблоны, снижающие производительность: «Наслоения хороши для торты, а не для программного обеспечения».

18 Реализация словарей Python: стремление быть всем во всем полезным

Эндрю Кучлинг (*Andrew Kuchling*)

В языке программирования Python словари являются основным типом данных.

Основные операции со словарем включают:

- добавление новой пары ключ–значение;
- извлечение значения, соответствующего определенному ключу;
- удаление существующих пар;
- последовательный перебор ключей, значений или пар ключ–значение.

Рассмотрим краткий пример использования словаря из командной строки Python-интерпретатора. (Чтобы попробовать этот пример в работе на Mac OS или на многих установках Linux, нужно лишь запустить команду `python`. Если Python еще не установлен, его можно загрузить с веб-сайта <http://www.python.org>.)

В следующем сеансе интерактивного общения группа символов `>>>` представляет собой приглашение к вводу интерпретатора Python, а `d` – это имя словаря, с которым я работаю:

```
>>> d = {1: 'January', 2: 'February',
... 'jan': 1, 'feb': 2, 'mar': 3}
{'jan': 1, 1: 'January', 2: 'February', 'mar': 3, 'feb': 2}
>>> d['jan'], d[1]
(1, 'January')
>>> d[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 12
>>> del d[2]
>>> for k, v in d.items(): print k,v # Перебор всех пар.
jan 1
1 January
mar 3
feb 2
```

В отношении такого типа данных, как словарь Python, нужно сделать два замечания.

- Отдельный словарь может содержать ключи и значения нескольких различных типов данных. Вполне допустимо в одном и том же словаре хранить ключи 1, 3+4j (комплексное число) и «abc» (строка). Значения сохраняют свои типы данных; они не конвертируются в строки.
- Ключи не имеют упорядоченной структуры. Такие методы, как `.values()`, которые возвращают все содержимое словаря, вернут данные в некотором произвольном порядке, не выстраивая их по значениям или по времени ввода.

Скорость выполнения операции извлечения ключей имеет весьма существенное значение, поэтому типы данных наподобие словарей зачастую реализуются в виде хэш-таблиц. Для реализации Python на языке Си (на которую впредь будут ссылки в виде CPython) словари являются даже более значимыми типами данных, потому что они поддерживают несколько других свойств языка. Например, классы и экземпляры классов используют словарь для хранения своих свойств:

```
>>> obj = MyClass() # Создание экземпляра класса
>>> obj.name = 'object' # Добавление свойства .name
>>> obj.id = 14 # добавление свойства .id
>>> obj.__dict__ # Извлечение основного словаря
{'name': 'object', 'id': 14}
>>> obj.__dict__['id'] = 12 # Сохранение в словаре нового значения
>>> obj.id # Соответствующее изменение свойства
12
```

Содержимое модуля также представлено словарем, наиболее характерный модуль `__builtins__` содержит такие встроенные идентификаторы, как `int` и `open`. Поэтому любые выражения, которые их используют, приводят к осуществлению ряда поисков по словарю. Другое использование словарей заключается в передаче ключевых аргументов в функции, поэтому словари потенциально могут создаваться и уничтожаться при каждом вызове функции. Такое внутреннее использование словарного типа данных означает, что каждый запуск программы на Python сопровождается одновременным существованием множества активных словарей, даже если код пользовательской программы не использует словари в явном виде. Поэтому очень важно, чтобы словари могли быстро создаваться и уничтожаться, не занимая слишком большой объем памяти.

Из реализации словарей в Python можно извлечь ряд уроков, касающихся кода, от которого зависит производительность работы.

Во-первых, следует поступиться преимуществами оптимизации и смириться с издержками, вызванными дополнительными расходами пространства памяти или времени вычисления. Были такие места, где разработчики Питона определили, что относительно простая реализация в конечном счете была лучше дополнительной оптимизации, которая поначалу казалась более привлекатель-

ной. Короче, за то, чтобы сохранять простоту вещей, зачастую приходится платить.

Во-вторых, главным критерием эффективности является реальная работа продукта; только таким способом можно узнать, что в действительности стоит делать.

Что делается внутри словаря

Словари представлены Си-структурой `PyDictObject`, определенной в файле `Include/dictobject.h`.

Вот как схематически выглядит структура, представляющая небольшой словарь, сопоставляющий «`aa`», «`bb`», «`cc`», ..., «`mm`» с целыми числами от 1 до 13:

```
int ma_fill 13
int ma_used 13
int ma_mask 31

PyDictEntry ma_table[]:
[0]: aa. 1 hash(aa) == -1549758592. -1549758592 & 31 = 0
[1]: ii. 9 hash(ii) == -1500461680. -1500461680 & 31 = 16
[2]: null. null
[3]: null. null
[4]: null. null
[5]: jj. 10 hash(jj) == 653184214. 653184214 & 31 = 22
[6]: bb. 2 hash(bb) == 603887302. 603887302 & 31 = 6
[7]: null. null
[8]: cc. 3 hash(cc) == -1537434360. -1537434360 & 31 = 8
[9]: null. null
[10]: dd. 4 hash(dd) == 616211530. 616211530 & 31 = 10
[11]: null. null
[12]: null. null
[13]: null. null
[14]: null. null
[15]: null. null
[16]: gg. 7 hash(gg) == -1512785904. -1512785904 & 31 = 16
[17]: ee. 5 hash(ee) == -1525110136. -1525110136 & 31 = 8
[18]: hh. 8 hash(hh) == 640859986. 640859986 & 31 = 18
[19]: null. null
[20]: null. null
[21]: kk. 11 hash(kk) == -1488137240. -1488137240 & 31 = 8
[22]: ff. 6 hash(ff) == 628535766. 628535766 & 31 = 22
[23]: null. null
[24]: null. null
[25]: null. null
[26]: null. null
[27]: null. null
[28]: null. null
[29]: ll. 12 hash(ll) == 665508394. 665508394 & 31 = 10
[30]: mm. 13 hash(mm) == -1475813016. -1475813016 & 31 = 8
[31]: null. null
```

Предфикс `ma_` в именах полей был взят из слова *mapping* (отображение), термина, использующегося в языке Python для типов данных, предоставляющих возможность поиска ключ–значение. Полями структуры являются:

`ma_used`

Количество слотов, занятых ключами (в данном случае – 13).

`ma_fill`

Количество слотов, занятых ключами или пустыми элементами (также 13).

`ma_mask`

Битовая маска, представляющая размер хэш-таблицы. Эта таблица содержит `ma_mask+1` слотов – в данном случае 32. Количество слотов в таблице всегда представлено числом в степени 2, поэтому значение всегда представлено в форме $2^n - 1$ для некоторого n и поэтому состоит из набора из n бит.

`ma_table`

Указатель на массив структур `PyDictEntry`, в котором содержатся указатели на:

- объект ключа;
- объект значения;
- кэшированную копию хэш-кода ключей.

Значение хэша кэшировано для ускорения работы. При поиске ключа точные значения хэша могут пройти быстрое сравнение, перед тем как будет выполнено довольно медленное полное сравнение ключей. Изменение размеров словаря также требует хэш-значения для каждого ключа, поэтому кэширование значения избавляет от необходимости заново помещать в хэш все ключи для проведения этой операции.

Мы не проводим непосредственное отслеживание количества слотов в таблице, а вместо этого при необходимости извлекаем это количество из `ma_mask`. При поиске элемента по ключу для определения исходного слота для конкретного значения хэша используется выражение `слот = хэш & маска`. Например, хэш-функция для первого элемента сгенерировала значение хэша $-1\ 549\ 758\ 592$, а $-1\ 549\ 758\ 592 \bmod 31$ – это 0, значит, элемент хранится в слоте 0.

Поскольку маска так часто востребована, мы храним ее вместо числа слотов. Количество слотов легко вычислить, прибавив 1, и нам никогда не приходится это делать в критичных для скорости фрагментах кода.

Поля `ma_fill` и `ma_used` обновляются при добавлении или удалении объектов. Поле `ma_used` – это количество ключей, имеющихся в словаре; добавление нового ключа увеличивает его на 1, а удаление ключа уменьшает его на 1. Чтобы удалить ключ, мы устанавливаем соответствующий указатель слота на пустой ключ; поэтому поле `ma_fill` при удалении ключа остается без изменений, но при добавлении нового ключа может быть увеличено на 1 (`ma_fill` никогда не уменьшается, но получает новое значение при изменении размеров словаря.)

Специальные приспособления

Стараясь быть полезным всем и во всем — предоставляя пользователям Python эффективный с точки зрения ресурсов времени и памяти тип данных, а разработчикам Python — внутреннюю структуру данных, используемую как часть реализации интерпретатора, и легко читаемую и обслуживаемую основу кода — приходится усложнять простую, теоретически элегантную реализацию, кодом для особых случаев, но главное при этом — не переусердствовать.

Особая оптимизация для небольших хэшей

Объект PyDictObject также содержит пространство хэш-таблицы из восьми слотов. В этой таблице могут храниться небольшие словари, состоящие из пяти и менее элементов, экономя время, затрачиваемое на дополнительный вызов `malloc()`. Это также улучшает использование кэш-памяти; к примеру, структуры PyDictObject при использовании x86 GCC занимают 124 байта памяти и поэтому могут помещаться в двух 64-байтных строках кэша. Словари, задействованные для параметров ключевых слов, чаще всего имеют от одного до трех ключей, поэтому данная оптимизация помогает улучшить производительность при вызове функций.

Когда специализация приводит к издержкам

Ранее мы выяснили, что отдельный словарь может содержать ключи нескольких различных типов данных. В большинстве программ на Python словари, являющиеся основой экземпляров класса и модулей, имеют в качестве ключей только строки. Вполне естественно возникает вопрос, обеспечивает ли преимущество специализированный словарный объект, который в качестве ключей допускает только строки. Может быть, специализированный тип данных будет полезнее и заставит интерпретатор работать быстрее?

Реализация на языке Java: еще одна специализированная оптимизация

На самом деле специализированный строковый словарный тип *существует* в Jython (<http://www.jython.org>), реализации Python в Java. Jython имеет класс `org.python.org.PyStringMap`, используемый только для словарей, в которых все ключи являются строками; он используется для словаря `_dict_`, являющегося основой экземпляров классов и модулей. Код Jython, создающий словари для пользовательского кода, применяет другой класс — `org.python.core.PyDictionary`, довольно тяжеловесный объект, который использует `java.util.Hashtable` для хранения своего содержимого и осуществления дополнительной косвенности, позволяющей `PyDictionary` иметь подклассы.

Определение языка Python не позволяет пользователям заменять внутренние словари `_dict_` другим типом данных и нести накладные расходы за счет ненужной поддержки создания подклассов. А для Jython наличие специализированного строкового словарного типа имеет определенный смысл.

Реализация на языке Си: динамический выбор функции хранения selecting the storage function dynamically

В CPython *нет* специализированного словарного типа, как в Jython. Вместо этого в нем используются различные ухищрения: отдельный словарь использует только строковую функцию, пока не потребуется поиск для нестроковых данных, а затем используется функция более широкого назначения. Все сделано довольно просто. `PyDictObject` содержит одно поле, `ma_lookup`, которое является указателем на функцию, используемую для поиска ключей:

```
struct PyDictObject {  
    ...  
    PyDictEntry *(*ma_lookup)(PyDictObject *mp, PyObject *key, long hash);  
};
```

`PyObject` — это Си-структура, которая представляет любой объект данных Python, содержащая основные поля, к которым относятся количество ссылок и указатель на объект типа. При необходимости структура может быть расширена за счет дополнительных полей такими специфическими типами, как `PyIntObject` и `PyStringObject`. Реализация словаря для поиска ключа вызывает `(dict->ma_lookup)(dict, key, hash)`; `key` — это указатель на `PyObject` представляет ключ, а `hash` — полученное для ключа хеш-значение.

Поле `ma_lookup` первоначально установлено на `lookdict_string`, функцию, для которой предполагается, что оба ключа — ключ в словаре и ключ, по которому идет поиск, — это строки, представленные стандартным типом Python — `PyStringObject`. Поэтому `lookdict_string` может принимать несколько сокращений. Одно из сокращений — построчное сравнение — никогда не вызывает исключений, поэтому часть ненужных проверок на ошибку может быть опущена. Также нет необходимости проводить проверку полного совпадения по объекту; произвольные типы данных Python могут предоставить свои собственные отдельные версии `<`, `>`, `<=`, `=` и `!=`, но стандартный строковый тип не имеет подобных исключений.

Как только будет обнаружен нестроковый ключ, то ли потому, что он использовался в качестве словарного ключа, то ли программа предприняла попытку провести по нему поиск, поле `ma_lookup` изменяется и указывает на более общую функцию `lookdict`. Функция `lookdict_string` проверяет тип входных данных и, если нужно, изменяет значение поля `ma_lookup`, а затем вызывает выбранную функцию, чтобы получить правильный ответ. (Дополнительно для CPython: это означает, что словарь, имеющий только строковые ключи, при запуске `d.get(1)` будет работать медленнее, даже если поиск не увенчается успехом. Весь последующий код программы, который ссылается на словарь, также будет

запущен через функцию более общего назначения и поэтому будет работать медленнее.) Подклассы `PyStringObject` должны рассматриваться не как строки, поскольку они могут определить новую проверку на равенство.

Конфликтные ситуации

Для любой реализации хэш-таблицы очень важно определить, что делать в том случае, если оба ключа выдают хэш, указывающий на один и тот же слот. Один из выходов — *связывание* (см. http://en.wikipedia.org/wiki/Hash_table#Chaining): каждый слот является заголовком связного списка, который содержит все элементы, по которым выдается хэш, указывающий на этот слот. Python не использует этот подход, поскольку создание связного списка потребует распределения памяти для каждого элемента этого списка, а это довольно медленные операции.

Следование за всеми указателями связного списка может также вызвать сокращение кэш-пространства. Альтернативный подход заключается в *открытой адресации* (см. http://en.wikipedia.org/wiki/Hash_table#Open_addressing): если первый испытуемый слот i не содержит ключа, то по заданному шаблону проходят апробацию другие слоты. Простейший шаблон называется *линейным апробированием*: если слот i занят, нужно опробовать $i+1$, $i+2$, $i+3$ и т. д., возвращаясь к слоту 0, когда будет достигнут конец таблицы. В Python линейное апробирование было бы слишком неэкономным, поскольку многие программы используют в качестве ключей последовательные целые числа, что приводит к возникновению блоков из заполненных слотов. Линейное апробирование приведет к частому сканированию этих блоков, снижая производительности работы. Вместо этого в Python используется более сложный шаблон:

```
/* Начальный слот */
slot = hash;

/* Начальное возмущающее значение */
perturb = hash;
while (<слот заполнен > && <элемент слота не равен ключу>) {
    slot = (5*slot) + 1 + perturb;
    perturb >= 5;
}
```

В коде на языке Си $5*slot$ записывается с использованием побитного смещения и сложения как $(slot \ll 2) + slot$. Фактор возмущения — $perturb$ имеет начальное значение, равное полному хэш-коду; затем его биты постепенно смещаются на пять битов вниз за один шаг. Это смещение гарантирует, что воздействие на каждый бит хэш-кода уже опробованного слота будет проведено достаточно быстро. Со временем фактор возмущения приобретет нулевое значение, и шаблон упростится до $slot = (5*slot) + 1$. В конечном счете будет генерированы все целые числа между нулем и ma_mask , поэтому поиск гарантированно со временем найдет либо ключ (при операции поиска), либо пустой слот (при операции вставки).

Значение смещения, равное 5 битам, было выбрано экспериментальным путем; 5 битов минимизируют конфликтные ситуации немного лучше, чем

4 или 6 битов, хотя эта разница не столь существенна. В ранних версиях кода использовались более сложные операции, в частности умножение или деление, но хотя эти версии обладали великолепной статистикой по числу конфликтных ситуаций, вычисление шло несколько медленнее.

(Более подробно история этой оптимизации изложена в исчерпывающих комментариях в файле `Objects/dictobject.c`.)

Изменение размера

При добавлении ключей разного размера словаря должен быть скорректирован. Код нацелен на поддержание заполнения таблицы на две трети; если словарь содержит n ключей, таблица должна иметь по крайней мере $n/(2/3)$ слотов. Этот коэффициент заполненности является компромиссом: более плотное заполнение таблицы приводит к возникновению дополнительных конфликтов при поиске ключа, но зато требует меньше памяти и поэтому легче помещается в кэш. Были предприняты эксперименты, в которых соотношение 2/3 корректировалось в зависимости от размера словаря, но они показали неудовлетворительные результаты; при каждой операции вставки требуется проверка необходимости расширения словаря, и те усложнения, которые возникают в операции вставки в связи с этой проверкой, ведут к замедлению работы.

Определение нового размера таблицы

Когда требуется скорректировать размер словаря, то как должен быть определен его новый размер? Для словарей небольшого и среднего размера, количество ключей в которых не превышает 50 000, новый размер составляет $ma_used * 4$. Многие написанные на Python программы, работающие с большими словарями, занимаются построением словаря в самом начале процесса выполнения, а затем ищут отдельные ключи или проходят в цикле по всему содержимому. Подобное учетверение размера словаря сохраняет его разреженность (коэффициент заполненности сначала составляет 1/4) и сокращает количество операций корректировки размера, осуществляемых в процессе его построения. Более объемные словари с количеством ключей, превышающим 50 000, используют $ma_used * 2$, чтобы избежать слишком больших расходов на пустые слоты.

При удалении ключа из словаря слот, который был занят ключом, изменяется и указывает на пустой ключ, а значение `ma_used` обновляется, но количество заполненных слотов в таблице не проверяется. Это означает, что словари при удалении ключа размер не изменяют. Если вы строите большой словарь, а затем удаляете из него множество ключей, словарная хеш-таблица может быть больше, чем при непосредственном создании словаря меньшего размера. Но такой пример использования случается нечасто. Из большинства мелких словарей, используемых для объектов и для передачи аргументов функций, ключи практически никогда не удаляются. Многие программы на Python создают словарь, работают с ним некоторое время, а затем «выбрасывают» его целиком. Поэтому вряд ли какая-нибудь программа на Python столкнется с использованием

последних резервов памяти из-за политики не изменять размеров при удалении.

Попеременное использование памяти: список свободных словарей

Многие экземпляры словарей используются самим языком Python для хранения параметров ключевых слов при вызове функций. Поэтому они создаются очень часто и имеют очень короткий срок существования, уничтожаясь после возврата из функции. Когда дело касается частых созданий и коротких жизненных циклов, эффективная оптимизация заключается в повторном использования незадействованных структур данных, сокращая тем самым количество вызовов `malloc()` и `free()`.

Поэтому Python содержит массив незадействованных словарных структур `free_dicts`. Длина этого массива в Python 2.5 состоит из 80 элементов. Когда требуется новый `PyDictObject`, указатель берется из `free_dicts`, и словарь задействуется снова. Словари добавляются к массиву, когда требуется их удаление; если `free_dicts` заполнен, структура просто освобождается.

Итерации и динамические изменения

В словарях часто используется циклический перебор содержимого. Методы `keys()`, `values()` и `items()` возвращают списки, в которых содержатся все имеющиеся в словаре ключи, значения или пары ключ–значение. Для экономии памяти пользователь может вызвать вместо них методы `iterkeys()`, `itervalues()` и `iteritems()`; они возвращают объект итератора, который возвращает элементы по одному. Но когда эти итераторы используются, Python, пока цикл не будет завершен, должен запретить работу любого оператора, который добавляет элемент в словарь или удаляет его оттуда.

Оказывается, эти ограничения ввести довольно просто. Итератор записывает количество элементов словаря при первом вызове метода `iter*()`. Если словарь изменяется, итератор вызывает исключение `RuntimeError` с сообщением «`dictionary changed size during iteration`» — изменение размеров словаря во время итерации.

Но есть один особый случай, при котором происходит модификация словаря в процессе работы кода перебора его элементов, когда задействуется код, присваивающий новое значение для того же самого ключа:

```
for k, v in d.iteritems():
    d[k] = d[k] + 1
```

Было бы уместным отменить вызов исключения `RuntimeError` при проведении подобных операций. Поэтому функция Си, обрабатывающая вставки значений в словари, `PyDict_SetItem()`, гарантирует сохранение размеров словаря, если вставляемый ключ в нем уже присутствует. Поисковые функции `lookdict()` и `lookdict_string` поддерживают это свойство, используя тот же способ, которым они выдают отчет об ошибке (если не найдут искомый ключ): при

неудачном поиске, они возвращают указатель на пустой слот, в котором будет сохранен искомый ключ. Это облегчает PyDict_SetItem сохранение нового значения в возвращенном слоте, который является либо пустым, либо слотом, о котором известно, что он занят тем же самым ключом. Когда новое значение записывается в слот, который уже занят таким же ключом, как в $d[k] = d[k] + 1$, словарный размер не проверяется на возможность проведения операции изменения размера, и RuntimeError отменяется. Поэтому код, подобный показанному в предыдущем примере, работает без вызова исключения.

Вывод

Несмотря на наличие многих свойств и параметров, предоставляемых словарями Python, и их довольно широкое внутреннее применение, реализация CPython до сих пор носит в основном слишком прямолинейный характер. Проведенная оптимизация является в значительной мере алгоритмической, и ее влияние на конфликтные показатели и на показатели тестирования производительности было проверено там, где это возможно, экспериментальным путем. Лучше всего углубить изучение реализации словарей, используя в качестве руководства исходный код. Для начала просмотрите файл Objects/dictnotes.txt, доступный по адресу <http://svn.python.org/view/python/trunk/Objects/dictnotes.txt?view=markup>, чтобы можно было всесторонне исследовать общие случаи использования словарей и всевозможные способы оптимизации. (Не все подходы, описанные в этом файле, были использованы в текущем коде.)

Затем прочтайте исходный текст в файле Objects/dictobject.c, который находится по адресу <http://svn.python.org/view/python/trunk/Objects/dictobject.c?view=markup>.

Вы сможете лучше разобраться с вопросами, читая комментарии и изредка просматривая код, чтобы получить соответствующее представление о его работе.

Благодарности

Спасибо Рэймонду Хеттингеру (Raymond Hettinger) за его комментарии к этой главе. Любые допущенные ошибки следует относить на мой счет.

19 Многомерные итераторы в NumPy

Трэйвис Олифант (*Travis E. Oliphant*)

NumPy – это пакет, расширяющий возможности языка Python за счет мощного объекта N-мерного массива – структуры данных, в которой для доступа к отдельным элементам используется N целых чисел, или индексов. Эта модель может быть успешно использована для обработки на компьютере широкого круга данных.

Например, в одномерном массиве могут храниться образцы звуков, в двухмерном массиве – изображения в серых тонах, в трехмерном массиве – цветные изображения (с одной из размерностей длиной в три или четыре элемента), а четырехмерный массив может хранить показатели звукового давления в зале во время концерта. Но полезными могут оказаться и более многомерные массивы.

NumPy обеспечивает среду для математических и структурных манипуляций с массивами произвольной размерности. Эти манипуляции лежат в основе многих научных, инженерных и мультимедийных программ, которые регулярно работают с большими объемами данных. Возможность осуществления этих математических и структурных манипуляций на языке высокого уровня может значительно упростить разработку и дальнейшее повторное использование этих алгоритмов.

NumPy предоставляет ассортимент математических вычислений, которые могут применяться к массивам наряду с обеспечением очень простого синтаксиса для структурных операций. В результате Python (с NumPy) может успешно применяться для разработки впечатляющих высокопроизводительных технических и научных программ.

Суть свойства, позволяющего проводить быстрые структурные манипуляции, заключается в том, что любая подобласть массива может быть выбрана за счет применения разрезания (*slicing*). В языке Python вырезка определяется начальным индексом, конечным индексом и шагом по индексу, для чего используется нотация `старт:стоп:шаг` (заключенная в квадратные скобки).

Представим, к примеру, что нам нужно сжать изображение из формата 656×498 в формат 160×120 за счет вырезки центральной области. Если изображение хранится в массиве NumPy под названием `im`, эта операция может быть выполнена с помощью следующей строки:

```
im2=im[8:-8:4, 9:-9:4]
```

Отличительной особенностью NumPy является то, что выбранное таким образом новое изображение использует общие данные с исходным изображением. Копия с исходного изображения не снимается. Эта оптимизация приобретает особую ценность при работе с большими наборами данных, когда беспорядочное копирование может растратить впустую вычислительные ресурсы.

Основные сложности обработки N-мерных массивов

В целях быстрого выполнения всех математических операций в NumPy реализованы циклы (на Си), способные проводить быструю обработку массива или нескольких массивов любой размерности. Создание универсального кода, способного быстро обрабатывать массивы произвольной размерности, может стать настоящей головоломкой. Возможно, создание цикла `for` для обработки всех элементов одномерного массива, или двух вложенных `for`-циклов для обработки всех элементов двумерного массива и не такая уж сложная задача. Действительно, если заранее известна размерность массива, для перебора всех его элементов можно воспользоваться точным количеством циклов `for`. Но как создать общий цикл `for`, который будет обрабатывать все элементы N-мерного массива, где N может быть произвольным целым числом?

У этой проблемы есть два основных решения. Одно из них заключается в использовании рекурсии, когда проблема рассматривается с точки зрения рекурсивных и общих составляющих. Соответственно, если `copy_ND(a, b, N)` — это функция, копирующая N-мерный массив, на который указывает `b`, в другой N-мерный массив, на который указывает `a`, то простая рекурсивная реализация может выглядеть следующим образом:

```
if (N==0)
    копирование памяти из b в a
    return
установка ptr_to_a и ptr_to_b
for n=0 до размера первого измерения a и b
    copy_ND(ptr_to_a, ptr_to_b, N-1)
    добавление шага_по_индексу_a[0] к ptr_to_a и шага_по_индексу_b[0]
    к ptr_b
```

Обратите внимание на то, что для всех случаев используется единственный цикл и общая проверка, которая останавливает рекурсию, когда значение N достигает нуля. Догадаться о том, как из любого алгоритма сделать рекурсивный, бывает очень сложно, даже если за основу будет взят только что показанный код. Рекурсия также требует использования вызова функции при каждом проходе цикла. Поэтому если при использовании рекурсии не пройти некоторую

оптимизацию общей составляющей (например выход из функции по $N=1$ и осуществление копирования памяти внутри цикла `for`), можно запросто получить довольно медленный код.

Большинство языков не станет делать такую оптимизацию в автоматическом режиме, поэтому рекурсивное решение, имеющее поначалу вполне элегантный вид, после добавления оптимизации с целью сокращения времени выполнения может выглядеть намного более замысловатым.

Вдобавок ко всему многие алгоритмы требуют хранения промежуточной информации, которая будет использована в ходе дальнейших рекурсий. Представьте, что будет, если понадобится отслеживать в массиве максимальное или минимальное значение? Обычно такие значения становятся частью структуры рекурсивного вызова и передаются в ней вместе с аргументами. В конечном счете, каждый алгоритм, который использует рекурсию, должен быть написан по-своему. Поэтому очень трудно предоставить программисту какой-нибудь дополнительный инструментарий, упрощающий его работу с рекурсивными решениями.

В NumPy для выполнения большинства алгоритмов работы с N-мерными массивами вместо рекурсий используются итерации. Любые рекурсивные решения могут быть переделаны в итеративные алгоритмы. Итераторы — это абстрактные понятия, упрощающие представление этих алгоритмов. Поэтому при использовании итераторов для обработки N-мерных массивов могут быть разработаны быстродействующие процедуры с легко читаемым и вполне понятным кодом, использующим всего одну циклическую структуру.

Итератор — это абстрактное понятие, в котором инкапсулируется идея последовательного перебора всех элементов массива с использованием всего одного цикла. В самом Python итераторы — это объекты, которые могут быть использованы в качестве предикатов любого цикла `for`. Например, код:

```
for x in iterobj:  
    process(x)
```

означает запуск функции `process` на обработку всех элементов `iterobj`. Главное требование к `iterobj` заключается в наличии в нем какого-то способа получения «следующего» элемента. Таким образом, понятие итератора перефокусирует проблему с выполнения цикла по всем элементам структуры данных на обнаружение следующего элемента.

Чтобы понять, как в NumPy реализованы и используются итераторы, важно иметь хотя бы некоторое представление о том, как NumPy просматривает память, занимаемую N-мерным массивом. Мы выясним этот вопрос в следующем разделе.

Модель памяти, занимаемой N-мерным массивом

Простейшая модель для N-мерного массива может быть применена в компьютерной памяти в том случае, если все элементы массива расположены друг за другом в последовательном сегменте. В таких условиях добраться до следую-

щего элемента массива так же просто, как добавить установленную константу к указателю на размещение в памяти текущих данных. В результате итератор для непрерывных массивов памяти требует только добавления установленной константы к текущему указателю на данные. Поэтому если N-мерный массив в NumPy был бы непрерывным, то обсуждение итераторов вряд ли представляло какой-нибудь интерес.

Красота абстракции итераторов заключается в том, что об управлении непоследовательными массивами и об их обработке можно думать с такой же легкостью, как и об обработке последовательных массивов. Массивы, состоящие из нескольких непоследовательных участков, получаются в NumPy из-за того, что массив может быть создан как «представление» нескольких других последовательных массивов памяти. Сами по себе эти новые массивы могут быть непоследовательными.

Рассмотрим, к примеру, трехмерный массив a , занимающий последовательный участок памяти. Используя NumPy, можно создать другой массив, состоящий из подмножества исходного, более крупного массива, воспользовавшись имеющейся в Python системой записи разрезания. Оператор $b=a[:, 2:3, 1::3]$ возвращает новый массив NumPy, состоящий из каждого второго элемента из первого измерения, всех элементов, начиная с четвертого (при нулевой базе индексации) из второго измерения, и каждого третьего элемента, начиная со второго элемента из третьего измерения. Этот новый массив не представлен копией памяти с указанных мест, а является представлением исходного массива и использует общую с ним память. Но вновь созданный массив не может быть представлен в виде непрерывного участка памяти.

Иллюстрация, использующая двумерный массив, должна закрепить в сознании это понятие. На рис. 19.1 показан последовательный, двумерный массив 4×5 , размещенный в блоках памяти, помеченных номерами от 1 до 20. Ниже представления массива 4×5 расположено линейное представление памяти, выделенной для массива, какой ее мог бы видеть компьютер. Если a представлен непрерывным участком памяти, то $b=a[1:3, 1:4]$ представлен закрашенной областью (участки памяти 7, 8, 9, 12, 13 и 14). Закрашенные места в линейном представлении показывают, что эти участки памяти не являются смежными.

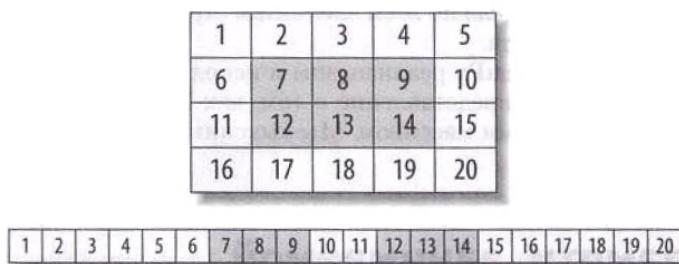


Рис. 19.1. Часть двумерного массива и его линейное представление в памяти

Основная модель памяти, использующаяся в NumPy для N-мерных массивов, поддерживает создание подобных непоследовательных представлений мас-

сивов. Это стало возможным благодаря тому, что к массиву прикреплен ряд целых чисел, представляющих значения для «пошагового перехода» по каждому измерению.

Значение шага для конкретного измерения определяет, сколько байтов должно быть пропущено, чтобы добраться от одного элемента массива до другого вдоль конкретного измерения или оси. Это значение шага может даже иметь отрицательное значение, указывающее на то, что следующий элемент массива может быть получен при движении по памяти в обратном направлении. Дальнейшее усложнение потенциально произвольного перехода по шагам будет означать, что построение итератора для управления общими случаями станет еще более трудной задачей.

Происхождение итератора NumPy

Циклы прохода по всем элементам массива в компилированном коде являются основной особенностью, предлагаемой NumPy программисту на языке Python. Применение итератора дает относительно простой и понятный способ создания этих циклов, которым можно воспользоваться для большинства основных (имеющих произвольные пошаговые переходы) циклов, поддерживаемых NumPy. Я считаю, что такая абстракция, как итератор, является собой пример красивого кода, поскольку дает возможность получить простое выражение простой идеи, даже при всей сложности основных подробностей реализации. Эта красота не возникает случайно, чаще всего она является результатом многочисленных попыток решения подборки схожих проблем, когда оформляется некое общее значение.

Моя первая попытка создания универсальной циклической конструкции для работы с N-измерениями была предпринята примерно в 1997 году, когда я стремился написать код как для N-мерного свертывания, так и для универсального отображения массива, позволяющего применить функцию Python для обработки каждого элемента N-мерного массива.

Решение, к которому я тогда пришел (хотя оно и не было оформлено как итератор), состояло в том, чтобы отслеживать целочисленный Си-массив, элементы которого использовались в качестве индексов для N-мерного массива. Итерация подразумевала увеличение этого N-индексного счетчика с помощью специального кода и перевода счетчика в нуль с увеличением следующего счетчика на единицу при достижении индексом размера массива в конкретном измерении.

Когда через восемь лет я создавал NumPy, я уже намного лучше разбирался как в самом понятии, так и в использовании объекта итератора в Python. Поэтому я стал рассматривать возможность подгонки кода массива-отображения под соответствующим образом оформленный итератор. В процессе работы я изучил, как Питер Вевреер (Peter Verveer, автор пакета `ndimage`, входящего в состав SciPy) добился N-мерного циклического перебора, и понял, что итератор был близок к тому, что я уже использовал. Уверившись в успехе, я formalizовал итератор, применив идеи, заложенные в `ndimage` к основным структурным

элементам, содержащимся в массиве-отображении, и к коду N-мерного свертывания.

Конструкция итератора

Как упоминалось ранее, итератор — это абстрактное понятие, в котором инкаспулирована идея последовательного перебора всех элементов массива. В NumPy используется следующий основной псевдокод цикла на основе итератора:

```
set up iterator          // настройка итератора
    (включая указание текущего значения на первое значение массива)
while iterator not done: // пока итератор не закончил работу
    process the current value // обработка текущего значения
    point the current value to the next value // установка текущего значения
                                                // указателя на следующее значение
```

Все, кроме обработки текущего значения, должно управляться итератором и заслуживает обсуждения. В конечном счете конструкция итератора состоит из трех основных частей:

1. механизма перемещения к следующему значению;
2. завершения работы;
3. настройки.

Все эти части будут рассмотрены отдельно. Положенные в основу итераторов NumPy конструкторские замыслы включали сокращение до минимума издержек от использования их внутри цикла и максимально возможное увеличение скорости их работы.

Перемещение итератора

Сначала нужно было принять решение о порядке извлечения элементов. Хотя можно было задумать итератор и без обеспечения какого-либо порядка извлечения элементов, все же в большинстве случаев программистам будет полезнее знать об этом порядке. Поэтому итераторы в NumPy следуют определенному порядку, который достигается за счет использования относительно простого подхода по типу обычновенного подсчета (с возвратом в исходное) с использованием кортежа чисел. Пусть кортеж из N целых чисел представляет текущую позицию в массиве, тогда $(0, \dots, 0)$ представляет первый элемент массива $n_1 \times n_2 \times \dots \times n_N$, а $(n_1 - 1, n_2 - 1, \dots, n_N - 1)$ представляет его последний элемент.

Кортеж целых чисел представляет N -разрядный счетчик. Следующая позиция получается увеличением последнего числа на единицу. Если в ходе этого процесса i -е число достигнет значения n_i , оно устанавливается в 0, а $(i-1)$ -е число увеличивается на 1.

Например, ведение счетчика для массива $3 \times 2 \times 4$ будет проходить следующим образом:

```
(0, 0, 0) (0, 0, 1) (0, 0, 2) (0, 0, 3) (0, 1, 0) (0, 1, 1) (0, 1, 2) (0, 1, 3) (1, 0, 0) ...
(2, 1, 2) (2, 1, 3)
```

Следующее увеличение на единицу приведет к показанию (0.0.0), и итератор будет установлен в позицию, с которой все начинается заново.

Этот счетчик является основой итератора NumPy. Способ, используемый для увеличения его значения, играет очень важную роль в его реализации. Поэтому реализация счетчика будет рассмотрена в следующем разделе. Предполагая, что нам уже доступен счетчик, определяющий правильную позицию в массиве, указатель на текущее значение в массиве всегда может быть получен умножением целых чисел счетчика на значения шага, определенные вместе с массивом, выдавая число байтов, добавляемых к адресу памяти первого элемента массива.

Например, если `data` – указатель на начальную позицию массива, `counter` – счетчик (или позиция) массива и `strides` – массив значений шага, то следующие операции:

```
currptr = (char *)data;
for (i=0; i<N; i++) currptr += counter[i]*strides[i];
```

устанавливают `currptr` на текущее значение массива (его первый байт).

На самом деле, если нужен указатель на текущее значение, вместо умножения реализация может отследить указатель одновременно с отслеживанием счетчика, внося корректировки при каждом изменении счетчика. Например, когда i -й индекс счетчика увеличивается на 1, `currptr` увеличивается на `strides[i]`. Когда i -й индекс переводится в 0, то по сути это то же самое, что и вычитание $n_i - 1$ из текущего индекса, и поэтому адрес памяти текущего значения массива должен быть уменьшен на $(n_i - 1) \times strides[i]$.

В общем, итератор поддерживает счетчик, определяя позицию в массиве вместе с указателем на текущее значение. В случае с массивом, у которого все элементы следуют в памяти друг за другом, сложение за этим счетчиком становится лишней нагрузкой, поскольку адрес памяти текущего значения массива, когда потребуется следующее значение, может обеспечиваться за счет увеличения значения адреса на размер каждого элемента в массиве.

Завершение работы итератора

Другим важным аспектом итератора (особенно если он используется в цикле) является определение момента завершения его работы и способа оповещения об этом. Наиболее распространенным подходом к оповещению является прикрепление к итератору переменной флагка, значение которой проверяется при каждом проходе цикла и устанавливается, когда элементы для итерации исчерпаны.

Одним из возможных способов установки флага может стать отслеживание перехода счетчика первого измерения из $n_1 - 1$ в 0. Проблема этого подхода в том, что ему нужна временная переменная для хранения последнего значения счетчика, поэтому он не работает с непрерывными массивами, для которых показания счетчика не отслеживаются.

Однако проще всего запомнить, какое конкретно число $(n_1 \times \dots \times n_N)$ итераций будет осуществляться при заданном размере массива. Это число может

быть сохранено в структуре итератора. Затем в ходе каждой итерации это число может уменьшаться. Когда оно достигнет 0, итератор должен завершить свою работу.

В NumPy используется незначительная модификация этого обратного отсчета. Чтобы сохранить общее число итераций в качестве нового информационного элемента, а также сохранить работающий счетчик проведенных к определенному моменту итераций, в NumPy используется целочисленный счетчик, с прямым отсчетом от нуля. Итерация завершается, когда показатель счетчика достигает общего количества элементов.

Настройка итератора

При создании итератора должен быть вычислен и сохранен размер массива, для которого он создается. Вдобавок к этому целочисленный счетчик должен быть установлен в 0, а счетчик позиции должен получить начальную установку ($0, 0, \dots, 0$). В заключение NumPy определяет, может ли итератор быть приспособлен для работы с простым непрерывным участком памяти, и устанавливает флагок, чтобы запомнить ответ.

Чтобы ускорить «шаг отслеживания возврата к началу», возникающий при переходе индекса счетчика от $n_i - 1$ к 0, результат $(n_i - 1) \times \text{strides}[i]$ заранее вычисляется и сохраняется для каждого индекса. Вдобавок к этому, чтобы избежать постоянно повторяющихся вычислений $n_i - 1$, эти значения также заранее вычисляются и сохраняются в структуре.

Хотя весьма сомнительно, что сохранение этого легко вычисляемого числа даст какой-нибудь прирост скорости, все же хранение размерности массива в структуре итератора принесет определенную пользу. Информацию о шагах также полезно хранить непосредственно в итераторе, вместе с переменной, в которой отслеживается количество измерений в обрабатываемом массиве. Когда размерности и шаги массива хранятся в итераторе, то последующие модификации трактовки массива могут быть легко выполнены путем изменения этих значений в итераторе, а не в самом исходном массиве. Это особеню полезно при передаче, которая превращает массивы, отличающиеся по формату, в массивы одного формата (что будет объяснено чуть позже).

И наконец, массив заранее вычисленных показателей сохраняется с целью упрощения вычислений, приводящих к однозначному соответствию единственного целочисленного счетчика массива и его N-индексного эквивалента. Например, на каждый элемент массива можно сослаться, используя единственный целочисленный указатель k , имеющий значения в диапазоне от 0 до $n_1 \times \dots \times n_{N-1} - 1$, или используя картеж целых чисел: (k_1, \dots, k_N) . Взаимосвязь может быть определена за счет $l_1 = k$, и:

$$k_i = \left\lfloor \frac{l_i}{\prod_{j=i+1}^N n_j} \right\rfloor,$$

$$l_i = l_{i-1} \bmod \left(\prod_{j=1}^N n_j \right).$$

А если наоборот, то взаимосвязь будет выражена следующим образом:

$$k = \sum_{i=1}^N k_i \left(\prod_{j=i+1}^N n_j \right).$$

В предыдущем уравнении выражения в скобках заранее вычислены и сохранены в итераторах в виде массива, как массив показателей, чтобы облегчить получение соответствий между двумя способами представления N-мерного индекса в обе стороны.

Определение показаний счетчика итератора

Код для отслеживания N-мерного индексного счетчика довольно прост. В нем должна быть учтена разница между простым добавлением 1 к последнему индексу и случаем возврата в исходное состояние. При каждом возврате могут возникать причины для возврата в исходное и других индексов. Поэтому для управления всеми остальными индексами должен быть какой-нибудь цикл `for`.

Проще всего было бы начинать с конца счетчика, или позиции массива, и двигаться в обратном направлении. На каждой индексной позиции код проверяет, не стала ли позиция меньше, чем $n_i - 1$. Если она достигла этого значения, к этой позиции просто добавляется 1 и к адресу памяти текущего значения указателя добавляется `strides[i]`. При каждом подобном случае может быть своевременно прерван цикл `for` и увеличено значение счетчика.

Если i -я позиция больше или равна $n_i - 1$, ее нужно сбросить в 0, а значение $(n_i - 1) \times strides[i]$ нужно вычесть из адреса памяти текущего значения указателя (чтобы вернуться к началу этого измерения). В этом случае проверена предыдущая индексная позиция.

Вся необходимая информация может быть представлена в вызываемой структуре со следующим содержимым:

`coords`

Позиционный индекс, N

`dims_m1`

Индекс для наивысшего элемента $n_i - 1$ каждого измерения

`strides`

Шаг каждого измерения

`backstrides`

Значение, позволяющее вернуться из конца каждого измерения в его начало: $(n_i - 1) \times strides[i]$

nd_m1

Количество измерений

currptr

Указатель на место в памяти для текущей позиции массива

Код для отслеживания показаний счетчика можно написать на Си:

```
for (i=it->nd_m1; i>=0; i--) {
    if (it->coords[i] < it->dims_m1[i]) {
        it->coords[i]++;
        it->dataptr += it->strides[i];
        break;
    }
    else {
        it->coords[i] = 0;
        it->dataptr -= it->backstrides[i];
    }
}
```

Этот вариант реализации использует оператор `break` и цикл `for`. Мы могли бы вместо этого воспользоваться оператором `while` и флагом, показывающим, нужно ли продолжать работу цикла:

```
done = 0;
i = it->nd_m1;
while (!done || i>=0) {
    if (it->coords[i] < it->dims_m1[i]) {
        it->coords[i]++;
        it->dataptr += it->strides[i];
        done = 1;
    }
    else {
        it->coords[i] = 0;
        it->dataptr -= it->backstrides[i];
    }
    i--;
}
```

Причина, по которой я выбрал цикл `for`, частично заключалась в том, что цикл `while` практически ничем от него не отличался (инициализация счетчика, проверка значения, увеличения значения счетчика). Обычно я приберегаю циклы `while` для случаев, когда итерации нуждаются в чем-то большем, чем единственный индекс итерации. И все же главная причина, по которой я выбрал цикл `for`, заключалась в том, что реализующий его фрагмент кода должен был использоваться в качестве макрокоманды для каждого цикла итерации. Я хотел избежать определения дополнительной переменной.

Структура итератора

Теперь настало время разобраться с полной структурой NumPy итератора. Она представлена следующей структурой на языке Си:

```
typedef struct {
    PyObject_HEAD
```

```

int nd_ml;
npy_intp index, size;
npy_intp coords[NPY_MAXDIMS];
npy_intp dims_ml[NPY_MAXDIMS];
npy_intp strides[NPY_MAXDIMS];
npy_intp backstrides[NPY_MAXDIMS];
npy_intp factors[NPY_MAXDIMS];
PyArrayObject *ao;
char *dataptr;
npy_bool contiguous;
} PyArrayIterObject;

```

Массивы этой структуры (`coords`, `dims_ml`, `strides`, `backstrides` и `factors`) имеют фиксированную длину с измерениями, управляемыми значением константы `NPY_MAXDIMS`. Этот вариант был выбран, чтобы упростить управление памятью. Тем не менее это не ограничивает числа используемых измерений. Этим легко можно управлять и по-другому, за счет динамического распределения требуемого объема памяти при создании итератора; такие изменения не отразятся на основном поведении.

Переменные `npy_intp` представлены целыми числами, размер которых достаточен для хранения указателя для платформы. Переменная `npy_bool` представляет собой флагок, который может принимать одно из значений — `TRUE` или `FALSE`. `PyObject_HEAD` — это часть структуры с содержимым, необходимым для всех объектов Python.

Ранее уже давались объяснения, касающиеся всех этих переменных, но чтобы прояснить этот вопрос, сделаем это еще раз:

`nd_ml`

Уменьшенное на единицу количество измерений массива: $N-1$.

`index`

Работающий счетчик, показывающий, на каком элементе массива в данный момент находится итератор. Этот счетчик получает значения от 0 до $size-1$.

`size`

Общее количество элементов массива: $n_1 \times n_2 \times \dots \times n_N$.

`coords`

- Массив из N целых чисел, обеспечивающий работу счетчика, или N -мерная позиция текущего элемента.

`dims_ml`

Массив из N целых чисел, предоставляющий уменьшенное на единицу число элементов в каждом измерении: $n_1 - 1, n_2 - 1, \dots, n_N - 1$.

`strides`

Массив из N целых чисел, предоставляющий количество байтов, которое нужно пропустить при переходе к следующему значению в конкретном измерении.

`backstrides`

Массив из N целых чисел, предоставляющий количество байтов, которое нужно вычесть, когда внутренний индексный счетчик конкретного измерения переходит от $n_i - 1$ к 0.

factors

Массив показателей, применяемых в ускоренных вычислениях соответствий между одномерным индексом и позициями N -мерного массива. Этот массив нужен только если осуществляется вызов PyArray_ITER_GOTO1D.

ao

Указатель на исходный массив, на основе которого построен этот итератор.

dataptr

Указатель на текущее значение массива (на его первый байт).

contiguous

Этот флагок принимает значение TRUE (1), если итератор используется для непрерывного массива, и FALSE (0) – в противном случае. То же самое, что и (ao->flags & NPY_C_CONTIGUOUS). Для непрерывного массива поиск следующего элемента осуществляется намного проще, поэтому его значение стоит проверить.

Интерфейс итератора

Итератор, реализованный в NumPy, использует комбинацию из макрокоманд и вызовов функций. Итератор создается Си-API вызовом функции `it=PyArray_IterNew(ao)`. Проверка завершения работы итератора может быть осуществлена с использованием макрокоманды `PyArray_ITER_NOTDONE(it)`. И наконец, переход на следующую позицию итератора осуществляется с помощью `PyArray_ITER_NEXT(it)`, макрокоманда, обеспечивающей встроенное выполнение (без вызова функции). В идеале эта макрокоманда должна быть встроенной функцией, потому что она имеет весьма сложное устройство. Однако поскольку NumPy написан на ANSI Си, в котором встроенные функции не определены, используется макрокоманда. И наконец, указатель на первый байт текущего значения может быть получен с использованием `PyArray_ITER_DATA(it)`, позволяющим избежать непосредственной ссылки на элемент структуры `dataptr` (с учетом будущего изменения имен элементов структуры).

Примером интерфейса итератора может послужить следующий кодовый фрагмент, вычисляющий наибольшее значение N -мерного массива. Предполагается, что массив называется `ao`, составлен из элементов типа `double` и правильно построен:

```
#include <float.h>
double *currval, maxval=-DBL_MAX;
PyObject *it;
it = PyArray_IterNew(ao);
while (PyArray_ITER_NOTDONE(it)) {
    currval = (double *)PyArray_ITER_DATA(it);
    if (*currval > maxval) maxval = *currval;
    PyArray_ITER_NEXT(it);
}
```

Этот код показывает, насколько относительно просто, используя структуру итератора, построить циклическую обработку элементов N -мерного массива.

состоящего из нескольких несмежных участков. Простота этого кода служит дополнительной иллюстрацией изящества абстракции итерации. Обратите внимание, насколько похож этот код на простой псевдокод итератора, показанный в начале предыдущего раздела «Конструкция итератора». Также учтите, что этот код работает с массивами произвольной размерности и произвольными шагами в каждом измерении, что не может не отразиться на вашей оценке красоты многомерного итератора.

Код, основанный на применении итератора, быстро работает как с непрерывными, так и с прерывающимися массивами. Но наиболее быстрая циклическая обработка непрерывного массива все же выглядит следующим образом:

```
double *currvil, maxval=-MAX_DOUBLE;
int size;
currvil = (double *)PyArray_DATA(ao);
size = PyArray_SIZE(ao);
while (size--) {
    if (*currvil > maxval) maxval = *currvil;
    currval += 1;
}
```

Реальное преимущество от использования NumPy итератора состоит в том, что он позволяет программистам создавать код, похожий на код обработки непрерывных массивов, который работает настолько быстро, что они даже не задумываются о том, являются ли их массивы непрерывными. Но нужно помнить, что принуждение к использованию алгоритма обработки непрерывных массивов тоже имеет свою цену, поскольку перед обработкой требует копирования прерывающихся данных в другой массив.

Разница в скорости работы между итератором NumPy и наиболее быстрым решением по обработке непрерывных массивов может быть существенно сокращена, если будет решена остающаяся проблема с текущим интерфейсом NumPy итератора. Суть этой проблемы в том, что макрокоманда `PyArray_ITER_NEXT` при каждом проходе цикла проводит проверку, не может ли итератор воспользоваться упрощенным подходом, предназначенным для работы с непрерывными массивами. Было бы идеально проводить такую проверку однократно, за пределами цикла, а затем внутри цикла для поиска следующего элемента использовать какой-нибудь один подход. Но такой тип интерфейса для реализации на Си немного не подходит. Для него потребуются две различные макрокоманды, похожие на `ITER_NEXT`, и два различных цикла `while`. В результате на момент написания этой главы в NumPy в этом плане так ничего и не было сделано. Предполагается, что тот, кто желает получить небольшой прирост скорости, доступный при обработке непрерывных массивов, имеет достаточную квалификацию для того, чтобы самостоятельно создать простой цикл (вообще отказавшись от итератора).

Использование итератора

Хорошая абстракция доказывает свою ценность, когда делает процесс программирования более простым при различных условиях или когда обнаруживается

польза от ее применения в тех сферах, для которых она первоначально не предназначалась. Для объекта итератора NumPy, безусловно, верны оба этих определения ценности. При использовании небольшой модификации простой исходный итератор NumPy становится рабочей лошадкой для реализации других свойств NumPy, среди которых итерация по всем, кроме одного измерения, и одновременная итерация по элементам нескольких массивов.

Кроме того, существование итератора и его расширений значительно упростило задачу быстрого добавления к коду некоторых усовершенствований, предназначенных для генерации случайных чисел и копирования массивов разной конфигурации.

Итерация по всем, кроме одного измерения

Основная идея в NumPy — получить прирост скорости, сконцентрировав оптимизацию на цикле по одному измерению, где можно применить простое пошаговое перемещение. Тогда используется стратегия итерации по всем, кроме последнего измерения. Такой подход был впервые представлен предшественником NumPy — Numeric для реализации математических функций.

В NumPy легкая модернизация итератора позволяет использовать основную стратегию в любом коде. Модифицированный итератор возвращается из следующего конструктора:

```
it = PyArray_IterAllButAxis(array, &dim).
```

Функция PyArray_IterAllButAxis принимает массив NumPy и адрес целого числа, представляющего измерение, удаляемое из итерации. Целое число передается по ссылке (оператор &), поскольку если указано измерение -1, функция определяет, какое измерение удалять, и помещает его номер в параметр. Когда входное измерение равно -1, подпрограмма выбирает измерение с наименьшим ненулевым шагом.

Для исключения может быть также выбрано измерение с наибольшим количеством элементов. Это позволит сократить до минимума количество итераций с внешним циклом и ограничит обработку большинства элементов предположительно более быстрым внутренним циклом. В этом случае проблема состоит в том, что получение информации в пределах и за пределами памяти зачастую является самой медленной частью алгоритма на процессорах общего назначения.

В результате в NumPy выбор пал на обеспечение работы внутреннего цикла с теми данными, которые расположены как можно ближе друг к другу. Очевидно, обращение к таким данным при проведении критичного по скорости работы внутреннего цикла будет происходить быстрее.

Итератор был модифицирован за счет следующих изменений.

1. Отделением длины удаляемого измерения от размера итерации.
2. Установкой количества элементов в выбранном измерении в 1 (таким образом, массив, в котором хранится на единицу меньше элементов, чем их общее количество, устанавливается в 0): `dims_m1[i]=0`.

- Установкой в измерении значения шага, необходимого для перехода от последнего к первому элементу в 0, чтобы постоянные возвраты счетчика в данном измерении к 0 никогда не изменяли указатель данных.
- Переустановкой флагка `contiguous` в 0, поскольку обработка не будет вестись в непрерывном участке памяти (каждая итерация должна пропускать целое измерение массива).

Измененный итератор возвращается функцией. Теперь его можно использовать везде, где раньше использовался обычный итератор. При каждом прохождении цикла итератор будет указывать на первый элемент избранного измерения массива.

Множественные итерации

Другой стандартной задачей в NumPy является согласованная итерация по нескольким массивам. Например, реализация сложения массивов требует итерации по обоим массивам с использованием связанного итератора, чтобы получающийся на выходе массив представлял в итоге произведение каждого элемента первого массива с каждым элементом второго. Этого можно достичь, используя особый итератор для каждого из входящих элементов и итератора для выходного массива, работающего по обычной схеме.

В качестве альтернативы NumPy предоставляет объект мультиитератора, способный упростить одновременную работу с несколькими итераторами. К тому же этот мультиитераторный объект автоматически работает с несколькими разными конфигурациями массивов (то есть реализует свойство распространяемости). Термин распространяемости (*broadcasting*) относится к свойству NumPy, позволяющему массивам с разными конфигурациями использовать вместе в операциях, поддерживающих поэлементную обработку. Распространяемость, к примеру, позволяет массиву с конфигурацией (4,1) быть добавленным к массиву с конфигурацией (3), в результате чего получается массив с конфигурацией (4,3). Распространяемость также позволяет проводить одновременную итерацию по массивам с конфигурациями (4,1), (3) и (5,1,1), получая при этом распространенную итерацию, охватывающую элементы массива с конфигурацией (5,4,3).

Свойство распространяемости подчиняется следующим правилам.

- Массивы с меньшим количеством измерений рассматриваются занимающими последние измерения массива с полным набором измерений, таким образом, все массивы получают полное число измерений. Новые создающиеся измерения заполняются единицами.
- Длина каждого измерения в итоговой распространенной конфигурации берется из самой большой длины этого измерения в любом из массивов.
- Для каждого измерения все исходные массивы должны иметь либо такое же количество элементов, как и получаемый в результате распространения результат, либо единицу в качестве числа элементов.

- Массивы с единственным элементом в конкретном измерении действуют, как будто этот элемент был фактически скопирован во все позиции во время итерации. Получается, что элемент «распространяется» по дополнительным позициям.

Ключ к реализации свойства распространяемости состоит из удивительно простых модификаций итераторов массива. После этих изменений стандартные циклы итератора могут напрямую использоваться для выполнения итоговых вычислений. К необходимым модификациям относятся изменения, вносимые в конфигурацию итератора (а не в исходный массив), и изменения в обычные шаги и в шаги возврата к первой позиции. Конфигурация, сохраняемая в итераторе, изменяется в соответствии с конфигурацией распространения. Шаги и возвратные шаги для распространяемых измерений изменяются до 0. При нулевом значении шага стандартный итератор фактически не перемещает указатель данных на элемент в памяти, используемый в качестве индекса этих нарощенных измерений. Тем самым желаемый эффект от свойства распространяемости получается без реального копирования в памяти.

Использование мультиитераторного объекта иллюстрируется следующим кодом:

```

PyObject *multi;
PyObject *in1, *in2;
double *i1p, *i2p, *op;
/* получение in1 и in2 (предполагается, что это массивы NPY_DOUBLE) */
/* первый аргумент - это число исходных массивов:
   следующие аргументы (количество которых может изменяться) - объекты
   массивов */
multi = PyArray_MultiNew(2, in1, in2);
/* создание выходного массива */
out = PyArray_SimpleNew(PyArray_MultiIter_NDIM(multi),
                        PyArray_MultiIter_DIMS(multi),
                        NPY_DOUBLE);
op = PyArray_DATA(out);
while(PyArray_MultiIter_NOTDONE(multi)) {
    /* получение указателей на текущие значения в каждом массиве */
    i1p = PyArray_MultiIter_DATA(multi, 0);
    i2p = PyArray_MultiIter_DATA(multi, 1);
    /* выполнение операции для данного элемента */
    *op = *i1p + *i2p
    op += 1; /* Продвижение указателя выходного массива */
    /* Продвижение всех входящих итераторов */
    PyArray_MultiIter_NEXT(multi);
}

```

Этот код очень похож на стандартный цикл итератора, за исключением того что мультиитератор приспосабливает входные итераторы под выполнение свойства распространяемости, а также осуществляет приращение всех других входных итераторов. Этот код управляет свойством распространяемости автоматически, в ходе обработки итератора, и сложение массива конфигурации (3,1) с массивом конфигурации (4) приводит к появлению выходного массива, имеющего конфигурацию (3,4).

Истории

Итераторы NumPy используются в коде NumPy повсюду, чтобы упростить структуру N -мерных циклов. Доступность итератора позволяет мне составлять код алгоритмов для самых распространенных массивов, состоящих из нескольких несмежных частей. Обычно трудности осмыслиения порядка обработки таких массивов наталкивали на вывод, что массивы нужно приводить к непрерывному виду (проводя необходимое для этого копирование), а затем использовать простой алгоритм циклической обработки. Существование итератора NumPy предоставило мне возможность создавать более универсальный код, сохраняющий хорошую читаемость кода и допускающий весьма незначительные потери в скорости работы с непрерывными массивами. Этот мелкий недостаток возмещается весьма существенным снижением требований к использованию памяти для прерывающихся массивов. Повышенная производительность при написании таких циклов является вполне достаточным аргументом в пользу существования итератора NumPy.

Но во всей красе польза абстракции проявляется при инкапсуляции в NumPy свойства распространяемости, в особенности когда мультиитератор дает мне возможность улучшить генераторы случайных чисел NumPy, чтобы можно было работать с массивами параметров, имеющих отношение к этим генераторам. На все эти изменения потребовалось около двух часов и лишь несколько строк кода.

Генератор случайных чисел NumPy был создан Робертом Керном (Robert Kern). Он не был знаком с API распространяемости на языке Си, который был добавлен незадолго до этого. В результате оригинальная реализация требовала, чтобы все параметры, использующиеся для определения случайных чисел, были скалярными значениями (например значение λ для экспоненциального распределения).

Это было весьма досадным ограничением. Ведь довольно часто возникает потребность в массиве случайных чисел, полученном из конкретного распределения, где у различных частей массива должны быть различные параметры. Например, программисту может понадобиться матрица случайных чисел, полученная из экспоненциального распределения, в которой каждая строка чисел должна быть выбрана с использованием различного значения λ . Чтобы получить возможность использования массива параметров, основные изменения касались использования мультиитераторного цикла (с его встроенным свойством распространяемости) и заполнения выходного массива экземплярами случайных чисел.

Другой благоприятный случай использования итератора появился, когда в код, копирующий данные из одного массива в другой, понадобилось внести изменения, позволяющие проводить копирование методом, согласующимся с определением имеющегося в NumPy свойства распространяемости. До этого массив копировался в новый массив с использованием стандартного итератора. Обеспечить однократное заполнение выходного массива можно было только проверкой его конфигурации. Если выходной массив выходил за пределы элементов, его итератор запускался снова. В конечном итоге стало понятно, что

такое поведение при копировании крайне нежелательно, поскольку при нем, по существу, выполнялся совершенно другой вид «распространяемости» (до тех пор пока общее число элементов одного массива было кратным числу элементов другого массива, любой массив мог быть скопирован в другой массив независимо от конфигурации). Подобный вид копирования данных, получаемый из этой команды копирования, не соответствовал определению распространяемости, используемому в других местах NumPy. Стало понятным, что он нуждается в изменении. И опять мультиитераторы и их встроенное понятие итераторной распространяемости послужили полезной абстракцией, потому что позволили мне написать код выполнения копирования (включающий проверку размера) очень быстро, с использованием всего лишь нескольких строк абсолютно нового кода.

Вывод

Имеющийся в NumPy объект итератора служит примером программной абстракции, упрощающей процесс программирования. Со временем его создания в 2005 году он проявил себя чрезвычайно полезным при написании N -мерных алгоритмов, работающих со всеми видами массивов NumPy, независимо от того, были они непрерывно расположены в памяти или фактически представляли собой прерывающиеся N -мерные участки, состоящие из каких-то непрерывных фрагментов памяти. В дополнение к этому незначительные модификации итератора значительно упростили реализацию некоторых более сложных идей NumPy, среди которых оптимизированная циклическая обработка (циклический перебор всех изменений, кроме измерения с наименьшим шагом) и распространяемость.

Итераторы являются весьма красивой абстракцией, поскольку они позволяют сэкономить драгоценное внимание программиста при реализации сложного алгоритма. То же самое можно сказать и о NumPy. Реализация в NumPy стандартного итератора массива позволила сделать процесс написания и отладки универсального кода намного более приятным и предоставила возможность инкапсуляции и проявления некоторых важных (но сложных в написании) внутренних свойств NumPy, к которым относится распространяемость.

20 Высоконадежная корпоративная система, разработанная для миссии NASA Mars Rover

Рональд Мак (Ronald Mak)

Часто ли вам приходится слышать, что красота находится в глазу наблюдателя? В нашем случае в качестве наблюдателя выступал марсоход NASA Mars Exploration Rover, и в отношении программного обеспечения этой миссии были очень жесткие требования по функциональности, надежности и устойчивости. Да, и еще, завершение разработки этого программного обеспечения должно было состояться точно по графику – Марс не принял бы оправданий за промахи в графике работ. Когда NASA говорит о выдерживании «окна запуска», то имеет в виду, что располагает не более чем одним вариантом!

В этой главе рассматривается проектирование и разработка совместного информационного портала – Collaborative Information Portal, или CIP, огромной корпоративной информационной системы, разработанной в NASA и использовавшейся руководством, инженерами и учеными миссии по всему миру.

Марсиане не терпят уродливого программного обеспечения. Для CIP понятие красоты не имело отношения к элегантности алгоритмов или программ, которыми можно было бы любоваться, слегка отступив назад. Красота, скорее всего, воплотилась в сложной структуре программного обеспечения, созданной мастерами своего дела, знающими, куда нужно заколачивать гвозди. Большие приложения могут порой обладать такой красотой, которую не часто встретишь в их более скромных собратьях. Здесь нужно отдать должное как более высоким требованиям, так и более существенным возможностям – большим приложениям зачастую приходится делать то, чего не нужно делать небольшим программам. Мы ознакомимся с основанной на языке Java комплексной архитектурой CIP, предназначеннной для обслуживания широкого круга запросов, а затем сфокусируем свое внимание на одной из ее служб как на предмете изучения и исследуем несколько фрагментов кода, изучив некоторые из тех «гвоздей».

которые позволяют системе отвечать предъявленным к ней требованиям функциональности, надежности и устойчивости.

Несложно предположить, что программное обеспечение, используемое в космических миссиях NASA, должно отличаться высокой надежностью. Эти миссии – слишком дорогое удовольствие, годы их проработки и миллионы долларов затрат не могут быть подвергнуты риску из-за несовершенства программного обеспечения. Разумеется, наиболее трудной частью работы над программным обеспечением является отладка и доводка программ, используемых на борту космического аппарата, которому предстоит быть в миллионах километров от Земли. Но надежность должна быть заложена и в наземные системы. Вряд ли кому-то захочется столкнуться с ошибкой в программе, из-за которой превратится ход миссии или будут потеряны ценные данные.

Говорить о красоте этого вида программного обеспечения можно только с некоторой долей иронии. В многоуровневой архитектуре, ориентированной на обслуживание широкого круга запросов, службы реализованы на среднем уровне, который находится на сервере. (Мы разработали все общие компоненты широкого использования уровня связующей программы, за счет чего существенно сократили время разработки.) Связующее программное обеспечение становится между клиентскими приложениями и выходными буферами источников данных; иными словами, приложению не нужно знать, где и как сохранены необходимые ему данные. Клиентское приложение совершаet удаленные служебные запросы к связующей программе, а затем получает ответы, в которых содержатся запрошенные данные. Когда все службы в связующей программе хорошо справляются со своей работой, конечные пользователи корпоративной системы даже и не знают, что их клиентские приложения делают удаленные служебные запросы. Когда связующая программа работает без сбоев, пользователи должны считать, что они обращаются к источнику данных напрямую и что вся обработка данных происходит на местном уровне, на их рабочих станциях или на ноутбуках. Таким образом, чем более удачным получается связующее программное обеспечение, тем менее заметным оно становится. Красивая связующая программа должна быть невидимой!

Миссия и совместный информационный портал

Главной целью миссии марсохода Mars Exploration Rover, или MER, был поиск следов воды, когда-либо протекавшей по поверхности Марса. В июне и июле 2003 года NASA запустила к Марсу два одинаковых марсохода, которые должны были действовать в качестве роботов-геологов. В январе 2004 года после раздельного семимесячного полета они сели на противоположные части планеты.

Каждый марсоход питался от солнечных батарей и мог самостоятельно передвигаться по поверхности. Каждый из них имел на борту научные приборы, среди которых были спектрометры, смонтированные на конце шарнирной руки. Эта рука оборудована сверлом и микроскопической телекамерой для изучения

того, что находится под каменистой поверхностью. Каждый марсоход оборудован несколькими камерами и антеннами для отправки данных и изображений на Землю (рис. 20.1).

Автоматические миссии NASA являются сочетанием аппаратного и программного обеспечения. Различные программные пакеты, находящиеся на борту каждого марсохода, осуществляют автономное управление и реагируют на удаленные команды, выдаваемые центром управления миссией из лаборатории реактивного движения NASA (Jet Propulsion Laboratory, JPL) недалеко от Пасадены, штат Калифорния. Программные пакеты, работающие на Земле в центре управления, дают возможность руководителям миссии, инженерам и ученым загружать и анализировать информацию, посланную марсоходами, планировать и разрабатывать новые командные последовательности для отправки на марсоходы и для сотрудничества с каждым из них.

В исследовательском центре NASA имени Эймса, недалеко от Маунтейн Вью, Калифорния, мы спроектировали и разработали совместный информационный портал (Collaborative Information Portal, CIP) для миссии MER. Команда проектировщиков состояла из десяти инженеров-программистов и специалистов по вычислительной технике. Среди остальных девяти членов команды были руководители проекта и представители службы поддержки, отвечающие за контроль качества, встраивание системы программного обеспечения, конфигурацию оборудования и задачи отслеживания дефектов.



Рис. 20.1. Марсоход (изображение любезно предоставлено JPL)

Потребности миссии

Мы спроектировали CIP для обеспечения трех первостепенных потребностей миссии MER. Для удовлетворения этих потребностей CIP обеспечивает персоналу управления миссией жизненно необходимое «знание складывающейся ситуации».

Управление временем

Успех любой большой и сложной миссии зависит от слаженных по времени действий всех занятых в ней людей, а миссия MER представляет на этот счет особые трудности. Поскольку персонал миссии работает по всему миру, CIP отображает время разных часовых поясов. Из-за того что марсоходы прибыли на противоположные стороны Марса, нужно еще учитывать и марсианские часовые пояса.

Изначально миссия работала по марсианскому времени, значит, все графики сеансов связи и событий (к примеру, загрузка данных с Марса) задавались по времени одного из марсианских часовых поясов, в зависимости от того, к какому марсоходу имел отношение сеанс или событие. Марсианский день примерно на 40 минут длиннее, чем земной, и поэтому относительно земного времени работа персонала миссии каждый день сдвигалась на более поздний срок соответственно этой разнице во времени, что отмечали их семьи и коллеги, живущие по земному времени. Это придало используемым в CIP функциям управления временем особую важность.

Управление персоналом

Другая жизненно важная функция была связана с отслеживанием работы персонала, что обуславливалось наличием двух команд управления марсоходами (по одной для каждого марсохода) и некоторыми специалистами, переходящими из команды в команду. CIP управляет списком персонала и отображает рабочие графики (в виде диаграмм Гантта), показывающие, кто, где и когда работает и что именно он делает.

CIP также облегчает организацию сотрудничества среди персонала миссии. Они могут рассыпать оповещения, совместно использовать анализы данных и изображения, выкладывать отчеты и комментировать отчеты друг друга.

Управление данными

Получение данных и изображений из далеких пределов вселенной является ключевой задачей каждой миссии NASA, относящейся к изучению планет и глубокого космоса, и здесь CIP также играет главную роль. Сложные наземные антенны решетки получают данные и изображения, посланные марсоходами, которые, попав на Землю, передаются в JPL для обработки и хранения на файловых серверах миссии.

Как только руководители миссии выпускают обработанные данные и файлы изображений, CIP генерирует метаданные, которые классифицируют файлы по различным критериям, среди которых: какой прибор марсохода сгенерировал данные, с какой камеры получено изображение, при каких установках, с использованием какой конфигурации, когда, где и т. п. После этого пользователи CIP могут искать данные и изображения по этим критериям и загружать их с файловых серверов миссии через Интернет на свои персональные ноутбуки и рабочие станции.

CIP также осуществляет защиту данных. К примеру, в зависимости от роли пользователя (и от того, является ли он гражданином США), могут быть наложены ограничения на конкретные данные и изображения.

Архитектура системы

Красота кода корпоративной системы складывается в том числе и из архитектуры, из того, как скомпонован программный код. Архитектура — это больше чем эстетика. В больших приложениях архитектура определяет порядок взаимодействия компонентов программного обеспечения и их вклад в общую надежность системы.

Мы создали СИР, используя трехуровневую архитектуру, предназначенную для обслуживания широкого круга запросов (service-oriented architecture, SOA). Мы придерживались промышленных стандартов, использовали передовой опыт, и там, где было возможно, использовали имеющееся в продаже готовое программное обеспечение (commercial off-the-shelf software, COTS). Программирование велось в основном на языке Java, мы использовали стандарты Java 2 Enterprise Edition (J2EE) (рис. 20.2).

Клиентский уровень в основном состоит из отдельных, имеющих графический интерфейс пользователя приложений на Java, выполненных с применением Swing-компонентов, и нескольких веб-приложений. J2EE-совместимый сервер прикладных программ работает на связующем уровне и является хост-узлом всех служб, отвечающих за обработку запросов, поступающих от клиентских приложений. При создании служб мы воспользовались технологией Enterprise JavaBeans (EJBs). Уровень данных состоит из источников данных и обслуживающих программ. Эти программы тоже созданы на Java и предназначены для управления файловым сервером, хранящим файлы обработанных данных и изображений. Обслуживающие программы генерируют метаданные для файлов, как только они будут выпущены руководителями миссии.

Использование SOA, основанного на J2EE, дало возможность воспользоваться вполне определенными bean-компонентами (и другими составляющими) везде, где это подходило для конструкции большого корпоративного приложения. Bean-компоненты сессии, не сохраняющие состояние (stateless session beans), управляют служебными запросами, не запоминая состояния при переходе от одного запроса к другому. С другой стороны, bean-компоненты сессии, запоминающие состояние, удерживают состояние информации для клиентов и, как правило, управляют сохраненной информацией, которую bean-компоненты считывают из хранилища данных и записывают в это хранилище. При разработке больших и сложных приложений важно иметь в запасе несколько вариантов для любой ситуации, возникающей в процессе проектирования.

В связующем программном обеспечении в качестве обслуживающих компонентов для каждой службы мы реализовали bean-компоненты сессии, не сохраняющие состояния. Это был фасадный метод, из которого мы генерировали веб-службу, использующуюся клиентскими приложениями для отправки запросов к службам и получения от них ответов. Каждая служба может также обращаться к одному и более bean-компонентам сессии, сохраняющим состояние, которые снабжаются всей необходимой логикой со стороны рабочих объектов и должны удерживать состояние между служебными запросами, к примеру, откуда считывать следующий блок информации из базы данных в ответ на запрос данных. В сущности, bean-компоненты, не сохраняющие состояние, зачастую служат в качестве служебных диспетчеров для bean-компонентов, сохраняющих состояние, которые и занимаются фактической работой.

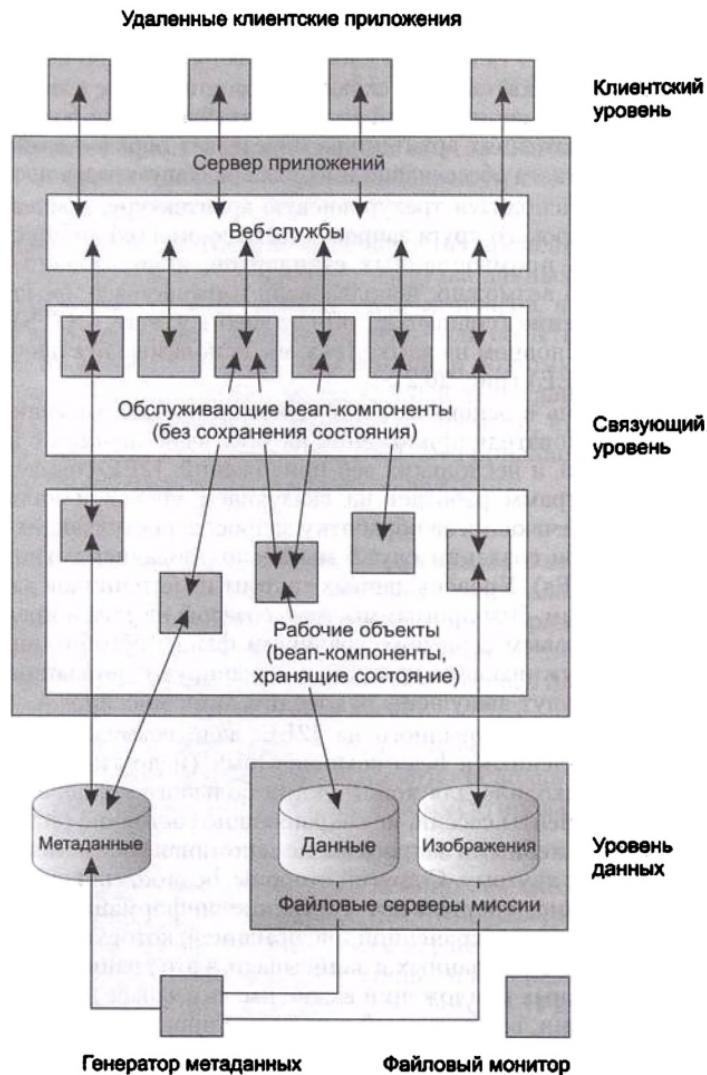


Рис. 20.2. Трехуровневая архитектура СИР, предназначенная для обслуживания широкого круга запросов

В архитектуре этого типа действительно можно найти много красивых моментов! В ней нашли воплощение некоторые ключевые принципы конструирования:

Опора на стандарты

Именно в исследовательском учреждении (таком как Исследовательский центр NASA имени Эймса, где мы спроектировали и разработали СИР) всегда есть сильное искушение изобрести кое-что новое, даже если это приведет

к повторному изобретению колеса. Миссия MER не предоставляла команде разработчиков СИР ни лишнего времени, ни ресурсов, и нашей целью было разработать для миссии код промышленного качества, для чего совсем не обязательно было вести какие-то исследования.

В любом большом приложении ключ к успеху заложен в интеграции, а не в написании программного кода. Красота, скрытая за соблюдением промышленных стандартов и использованием передового опыта, раскрывается в том, что мы уменьшаем объем программирования за счет использования COTS-компонентов, которые благодаря общим интерфейсам способны слаженно работать друг с другом. Это позволило нам дать руководителям миссии гарантии поставки в срок полноценного и надежного кода.

Слабая взаимосвязанность

Мы установили слабую взаимосвязанность между клиентскими приложениями и службами связующего программного обеспечения. Это означало, что как только программист, разрабатывающий приложение, и программист, разрабатывающий службу, договаривались об интерфейсе, они могли разрабатывать соответственный код параллельно. Любые изменения, вносимые одной стороной, не влияли на другую сторону, пока интерфейс сохранялся неизменным. Слабая взаимосвязанность была другим главным фактором, позволяющим нам завершить в срок разработку большого многоуровневого SOA-приложения.

Независимость от языка программирования

Для связи друг с другом клиентские приложения и сервисы, поддерживаемые связующим программным обеспечением, использовали веб-службы. Протокол веб-служб является промышленным стандартом, обладающим языковой независимостью. Большинство клиентских СИР-приложений были написаны на Java, но связующее программное обеспечение занималось обслуживанием некоторых приложений, написанных на C++ и C#. Как только нам удалось добиться от сервиса работы с Java-клиентом, заставить его работать с другими клиентами, поддерживающими веб-службы и написанными на любых других языках программирования, было относительно простой задачей. Это существенно увеличило полноценность и удобство использования СИР при минимальных дополнительных затратах времени и ресурсов.

Модульность

С ростом размера приложения важность модульного подхода возрастает многократно. Каждая служба в СИР является самостоятельным связующим компонентом, независимым от других служб. Если одной службе нужно работать с другой, она делает служебный запрос к другой службе, как если бы она была клиентским приложением. Это позволило нам создавать службы по отдельности и добавить иное измерение параллельности нашей разработке. Модульные службы — это красивые образцы, которые часто можно найти в больших, успешно работающих SOA-приложениях.

На клиентском уровне прикладные программы часто комбинируют обращения к службам, либо объединяют результаты нескольких служб, либо используют результаты одной службы для передачи их в запросе к другой службе.

Расширяемость

Использование уровней CIP оживляется, как только управление миссией выпускает обработанные данные и файлы изображений, особенно после того как одним из марсоходов будет обнаружено что-нибудь интересное. Нам нужно было гарантировать, что связующее программное обеспечение CIP сможет справиться с подобными всплесками активности, когда у пользователей появится сильное желание загрузить и просмотреть самые последние файлы. В такие моменты замедление работы или, что еще хуже, возникновение аварийной ситуации было бы абсолютно неприемлемым и очень заметным.

Красота инфраструктуры J2EE, в частности, состоит в том, как она справляется с расширяемостью. Сервер приложений поддерживает объединения bean-компонентов, и, в зависимости от потребностей, он может автоматически создавать больше экземпляров поставляющих услуги bean-компонентов сессий, не сохраняющих состояния. Этой замечательной «бесплатной» особенностью J2EE разработчики службы связующего уровня не преминули воспользоваться.

Надежность

Будучи стандартом, подвергшимся существенной промышленной проверке, инфраструктура J2EE доказала свою исключительную надежность. Мы не стали расширять границы ее предназначений, определенные разработчиками. Итак, после двух лет эксплуатации CIP поставил рекорд непрерывной работы, который превысил 99,9 %.

Мы превысили показатель надежности, фактически предусмотренный для J2EE. На следующей стадии изучения проблемы вы увидите, что мы забили несколько дополнительных «гвоздей» в наши службы для того, чтобы поднять их надежность еще выше.

Исследование конкретного примера: Служба информационных потоков

Мы уже посмотрели на красоту CIP, заложенную на уровне архитектуры. Теперь настало время сосредоточить внимание на одной из его связующих служб – на службе информационных потоков как на конкретном поучительном примере и изучить некоторые «гвозди», позволившие нам удовлетворить требования по обеспечению миссии работоспособным, надежным и имеющим достаточный запас прочности программным обеспечением. Вы увидите, что эти «гвозди» не отличались особой затейливостью; красота заключалась в знании тех мест, в которые их нужно было заколачивать.

Функциональное назначение

Одна из задач управления данными миссии MER состояла в том, чтобы дать возможность пользователям загружать файлы данных и изображений с файло-

вого сервера миссии, размещенного в лаборатории JPL, на их персональные рабочие станции и ноутбуки. Как уже было рассказано, на уровне данных утилиты CIP генерировали метаданные, позволявшие пользователям искать нужные файлы по различным критериям. Пользователям также нужна была возможность выкладывать на серверы свои аналитические отчеты.

Служба информационных потоков CIP выполняет загрузку и выкладывание файлов. Мы дали этой службе такое название, потому что она обеспечивает безопасный поток файлов через Интернет между файловыми серверами миссии, расположенными в JPL, и локальными компьютерами пользователей. В ней используются протоколы веб-служб, поэтому клиентские приложения могут быть написаны на любом языке, поддерживающем протокол, и для этих приложений можно свободно разрабатывать любые приемлемые графические интерфейсы.

Удаленные клиентские приложения

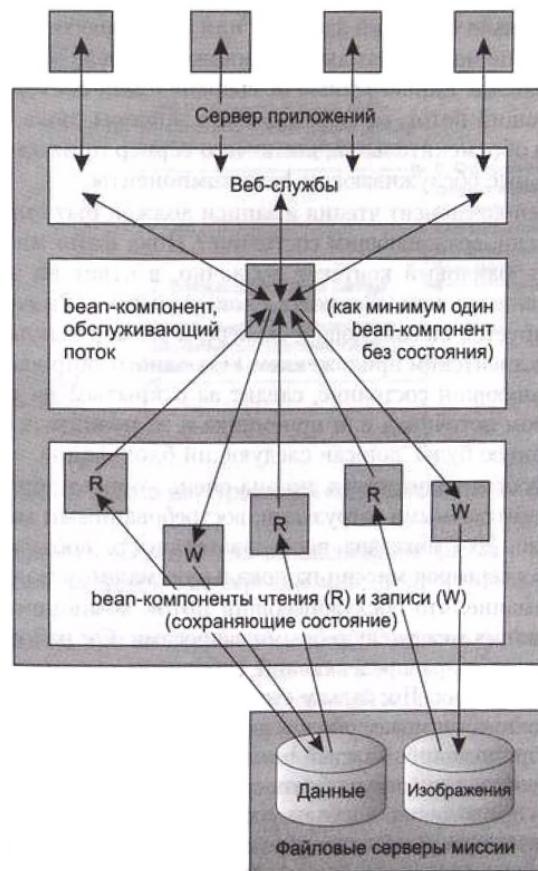


Рис. 20.3. Архитектура службы информационных потоков

Архитектура службы

Подобно всем другим связующим службам, потоковая служба для получения клиентских запросов и возврата ответов использует веб-службы. Каждый запрос сначала выставляется обслуживающим компонентом потока, который реализован bean-компонентом сессии, не сохраняющим состояние. Обслуживающий компонент создает компонент чтения файла, реализованный в виде сохраняющего состояния bean-компонента сессии, предназначенного для совершения действий по загрузке запрошенного файлового содержимого на машину клиента. В обратном направлении обслуживающий компонент создает компонент записи файла, который также реализован в виде сохраняющего состояния bean-компонента сессии, предназначенного для выкладывания файлового содержимого на сервер (рис. 20.3).

В любой момент несколько пользователей могут загружать или выкладывать файлы, и каждый отдельный пользователь может также одновременно производить несколько операций загрузки или выкладывания файлов. То есть на связующем уровне может быть активно множество bean-компонентов считывания и записи файлов. Единственный не сохраняющий состояния bean-компонент, обслуживающий поток, обрабатывает все запросы, пока нагрузка не становится для него обременительной, после чего сервер приложений может создать дополнительные обслуживающие bean-компоненты.

Почему каждый компонент чтения и записи должен быть представлен bean-компонентом сессии, сохраняющим состояние? Пока файл маленький, служба потока переносит файловый контекст поблочно, в ответ на запросы клиента «Считать блок данных» или «Записать блок данных». (Размер загружаемого блока конфигурируется на связующем сервере, а размер выкладываемого файла определяется клиентским приложением.) От одного запроса к другому bean-компонент, сохраняющий состояние, следит за открытым на файловых серверах миссии файлом источника или приемника и за позицией, с которой будет считан или в которую будет записан следующий блок файла.

Это весьма простая архитектура, но она очень хорошо справляется с одновременными множественными загрузками, востребованными множеством пользователей. На рис. 20.4 показана последовательность событий при загрузке файла с файловых серверов миссии на локальную машину пользователя.

Обратите внимание, что обслуживающий поток компонент не удерживает какого-либо состояния между служебными запросами. Он работает как скоростной диспетчер службы, распределяющий работу по сохраняющим состояния bean-компонентам чтения. Поскольку ему не нужно отслеживать запросы или удерживать состояние, он может обслуживать запросы вперемешку от нескольких клиентских приложений. Каждый bean-компонент чтения файла удерживает информацию состояния (откуда брать следующий блок данных) для отдельного клиентского приложения, поскольку для загрузки всего файла приложение создает несколько запросов «Считать блок файла». Эта архитектура позволяет службе потоков загружать несколько файлов для нескольких клиентов одновременно, предоставляя для всех вполне приемлемую пропускную способность.

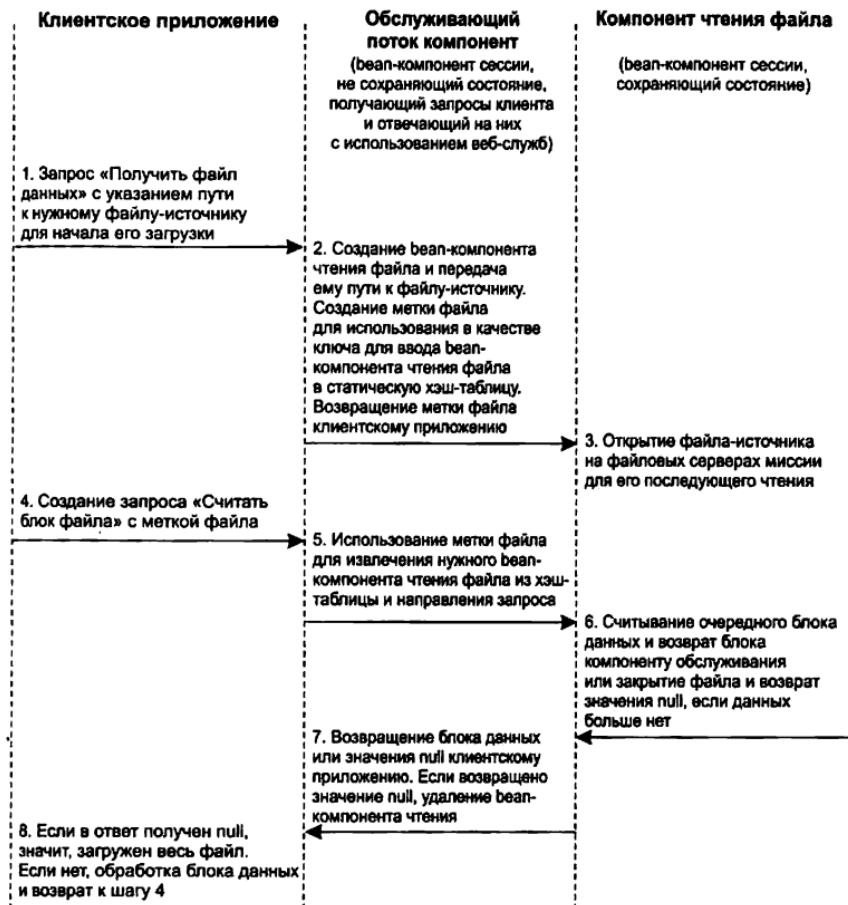


Рис. 20.4. Работа двухуровневой службы при чтении файла

Последовательность событий для выкладывания файла с пользовательской локальной машины на файловые серверы миссии проста и понятна. Она показана на рис. 20.5.

Хотя в таблицах это и не отражено, наряду с меткой файла каждый запрос имеет также и метку пользователя. Сначала клиентское приложение получает метку пользователя, когда проходит удачный запрос на вход в систему (в котором используется имя пользователя и пароль) к службе обслуживания пользователей на связующем уровне, с помощью которого осуществляется аутентификация пользователя. Метка пользователя содержит информацию, которая идентифицирует конкретную сессию пользователя, включая его роль. Это позволяет службе потоков осуществлять проверку полномочий пользователя на выдачу запроса. Она проверяет роль пользователя, чтобы убедиться, что у него есть права на загрузку конкретного файла. Например, миссия MER не

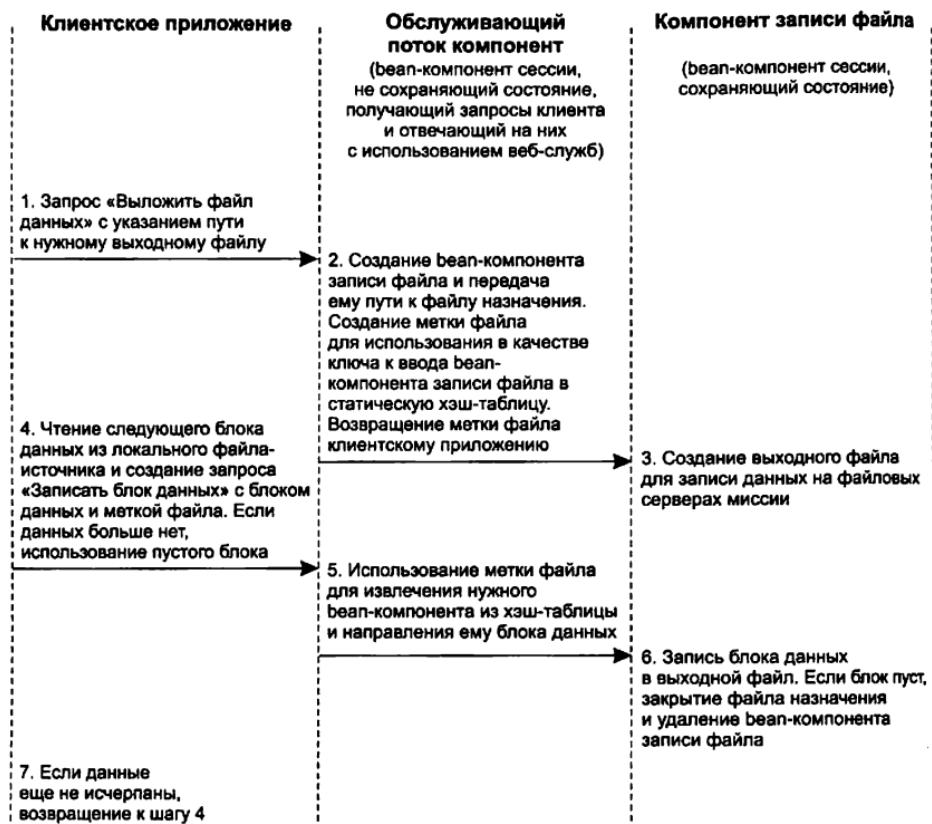


Рис. 20.5. Работа двухуровневой службы при записи

разрешает некоторым заграничным пользователям (не являющимся гражданами США) доступ к конкретным файлам, и CIP соблюдает все подобные ограничения по защите информации.

Надежность

Надежный программный код сохраняет постоянную работоспособность, не создавая при этом никаких проблем. Сбои случаются крайне редко, если вообще когда-нибудь происходят. Нетрудно представить, что код, работающий на борту марсохода, должен быть абсолютно надежен, поскольку потребности в его поддержке удовлетворить не так-то просто. Но для миссии MER требовалось, чтобы и наземное программное обеспечение также было надежным. Как только миссия вошла в стадию осуществления, никто не хотел сталкиваться с проблемами программного обеспечения, способными сорвать какие-нибудь операции.

Я уже упоминал, что при разработке проекта CIP был предпринят ряд мер для обеспечения внутренней надежности системы.

- Соблюдение промышленных стандартов и использование передового опыта, включая использование J2EE.
- Использование хорошо себя зарекомендовавшего готового программного обеспечения везде, где это возможно, включая промышленный сервер приложений от общепризнанного поставщика связующего программного обеспечения.
- Использование архитектуры, ориентирующейся на службы, выполненные в виде отдельных модулей.
- Реализация простых и понятных связующих служб.

Мы улучшили показатели надежности за счет некоторых дополнительных «гвоздей»: регистрации работы службы и отслеживания процесса работы. Эти возможности могут быть полезны и при отладке небольших программ, но они становятся жизненно необходимыми при отслеживании рабочего поведения больших приложений.

Ведение регистрационного журнала

В процессе разработки для регистрации практически всего, что происходит в связующих службах, мы воспользовались Java-пакетом с открытым кодом Apache Log4J. Он, конечно, пригодился и для отладки программ в процессе их создания. Ведение регистрационного журнала позволило нам создать более надежный код. Как только появлялась ошибка, регистрационный журнал сообщал нам, что происходило непосредственно перед возникновением проблемы, благодаря чему мы получали больше возможностей для устранения ошибки.

Сначала, еще до ввода в строй портала CIP, мы хотели ограничить регистрацию, оставив только самые важные сообщения. Но все же остановились на том, что надо оставить весь основной объем регистрации, поскольку влияние на общую производительность было незначительным. Потом мы обнаружили, что регистрация дает нам много полезной информации, не только о том, что происходит с каждой службой, но также и о том, как клиентские приложения используют службы. Благодаря анализу регистрационных журналов (которые мы назвали «регистрационными раскопками») мы смогли произвести тонкую настройку служб и добиться от них лучшей производительности, основываясь на данных, полученных опытным путем (см. далее раздел «Динамическая реконфигурация»).

Вот некоторые фрагменты кода от bean-компоненты, обслуживающего потоки, которые показывают, как мы реализовали регистрацию загрузки файла. Метод `getDataFile()` обрабатывает каждый запрос «Получить файл данных» (используя веб-службы), поступающий от клиентских приложений. Метод немедленно регистрирует запрос (строки 15–17), включая в запись идентификатор пользователя, выдавшего запрос, и путь к востребованному файлу-источнику:

```

1 public class StreamerServiceBean implements SessionBean
2 {
3     static {
4         Globals.loadResources("Streamer");
5     };
6
7     private static Hashtable readerTable = new Hashtable();
8     private static Hashtable writerTable = new Hashtable();
9
10    private static BeanCacheStats cacheStats = Globals.queryStats;
11
12    public FileToken getDataFile(AccessToken accessToken, String filePath)
13        throws MiddlewareException
14    {
15        Globals.streamerLogger.info(accessToken.userId() +
16            ": Streamer.getDataFile(" +
17            + filePath + ")");
18        long startTime = System.currentTimeMillis();
19
20        UserSessionObject.validateToken(accessToken);
21        FileToken fileToken = doFileDownload(accessToken, filePath);
22        cacheStats.incrementTotalServerResponseTime(startTime);
23        return fileToken;
24    }
25

```

Метод `doFileDownload()` создает новую метку файла (строка 30) и bean-компонент чтения файла (строка 41), а затем вызывает метод этого компонента `getDataFile()` (строка 42). Поле `cacheStats` работает с мониторингом времени выполнения, который будет рассмотрен позже:

```

26    private static FileToken doFileDownload(AccessToken accessToken,
27                                            String filePath)
28        throws MiddlewareException
29    {
30        FileToken fileToken = new FileToken(accessToken, filePath);
31        String key           = fileToken.getKey();
32
33        FileReaderLocal reader = null;
34        synchronized (readerTable) {
35            reader = (FileReaderLocal) readerTable.get(key);
36        }
37
38        // Создание bean-компонента чтения для начала загрузки.
39        if (reader == null) {
40            try {
41                reader = register.NewReader(key);
42                reader.getDataFile(filePath);
43
44                return fileToken;
45            }
46            catch(Exception ex) {
47                Globals.streamerLogger.warn("Streamer.doFileDownload(" +
48                    + filePath + "): " +
49                    ex.getMessage());
50                cacheStats.incrementFileErrorCount();

```

```

51         removeReader(key, reader);
52     }
53   }
54 } else {
55     throw new MiddlewareException("File already being downloaded: " +
56                                     filePath);
57 }
58 }
59 }
60

```

Метод `readDataBlock()` обрабатывает каждый запрос «Считать блок данных», поступивший от клиентского приложения. Он ищет нужный bean-компонент чтения файла (строка 71) и вызывает его метод `readDataBlock()` (строка 79). Когда файл-источник закончится, он удаляет bean-компонент чтения файла (строка 91):

```

61 public DataBlock readDataBlock(AccessToken accessToken, FileToken
62     fileToken) throws MiddlewareException
63 {
64     long startTime = System.currentTimeMillis();
65     UserSessionObject.validateToken(accessToken);
66
67     String key = fileToken.getKey();
68
69     FileReaderLocal reader = null;
70     synchronized (readerTable) {
71         reader = (FileReaderLocal) readerTable.get(key);
72     }
73
74     // Использование bean-чтения для загрузки следующего блока данных.
75     if (reader != null) {
76         DataBlock block = null;
77
78         try {
79             block = reader.readDataBlock();
80         }
81         catch(MiddlewareException ex) {
82             Globals.streamerLogger.error("Streamer.readDataBlock(" +
83                                         + key + ")". ex);
84             cacheStats.incrementFileErrorCount();
85             removeReader(key, reader);
86             throw ex;
87         }
88
89         // Конец файла?
90         if (block == null) {
91             removeReader(key, reader);
92         }
93
94         cacheStats.incrementTotalServerResponseTime(startTime);
95         return block;
96     }
97     else {
98         throw new MiddlewareException(
99             "Download source file not opened: " +

```

```

100         fileToken.getFilePath());
101     }
102 }
103

```

Методы `register.NewReader()` и `removeReader()` соответственно создают и уничтожают сохраняющие состояние bean-компоненты чтения:

```

104 private static FileReaderLocal register.NewReader(String key)
105     throws Exception
106 {
107     Context context = MiddlewareUtility.getInitialContext();
108     Object queryRef = context.lookup("FileReaderLocal");
109
110    // создание служебного bean-чтения и его регистрация.
111    FileReaderLocalHome home = (FileReaderLocalHome)
112        PortableRemoteObject.narrow(queryRef, FileReaderLocalHome.class);
113    FileReaderLocal reader = home.create();
114
115    synchronized (readerTable) {
116        readerTable.put(key, reader);
117    }
118
119    return reader;
120 }
121
122 private static void removeReader(String key, FileReaderLocal reader)
123 {
124    synchronized (readerTable) {
125        readerTable.remove(key);
126    }
127
128    if (reader != null) {
129        try {
130            reader.remove();
131        }
132        catch(javax.ejb.NoSuchObjectLocalException ex) {
133            // игнорирование
134        }
135        catch(Exception ex) {
136            Globals.streamerLogger.error("Streamer.removeReader(" +
137                                         + key + ")", ex);
138            cacheStats.incrementFileErrorCount();
139        }
140    }
141 }
142 }

```

Теперь рассмотрим фрагменты кода из bean-компонента чтения файла. Поля `cacheStats` и `fileStats` предназначены для мониторинга времени выполнения, который будет рассмотрен позже. Метод `getDataFile()` регистрирует начало загрузки файла (строки 160–161):

```

143 public class FileReaderBean implements SessionBean
144 {
145     private static final String FILE = "file";
146

```

```
147 private transient static BeanCacheStats cacheStats = Globals.queryStats;
148 private transient static FileStats fileStats = Globals.fileStats;
149
150 private transient int totalSize;
151 private transient String type;
152 private transient String name;
153 private transient FileInputStream fileInputStream;
154 private transient BufferedInputStream inputStream;
155 private transient boolean sawEnd;
156
157 public void getDataFile(String filePath)
158     throws MiddlewareException
159 {
160     Globals.streamerLogger.debug("Begin download of file " +
161                             + filePath + "'");
162     this.type = FILE;
163     this.name = filePath;
164     this.sawEnd = false;
165
166     try {
167
168         // Создание входного потока из файла данных.
169         fileInputStream = new FileInputStream(new File(filePath));
170         inputStream = new BufferedInputStream(fileInputStream);
171
172         fileStats.startDownload(this, FILE, name);
173     }
174     catch(Exception ex) {
175         close();
176         throw new MiddlewareException(ex);
177     }
178 }
179 }
```

Метод `readDataBlock()` считывает каждый блок данных из файла-источника. Когда весь файл будет считан, он регистрирует завершение работы (строки 191–193):

```
180     public DataBlock readDataBlock()
181         throws MiddlewareException
182     {
183         byte buffer[] = new byte[Globals.streamerBlockSize];
184
185         try {
186             int size = inputStream.read(buffer);
187
188             if (size == -1) {
189                 close();
190
191                 Globals.streamerLogger.debug("Completed download of " +
192                             + type + " " + name + ": " +
193                             totalSize + " bytes");
194
195                 cacheStats.incrementFileDownloadedCount();
196                 cacheStats.incrementFileByteCount(totalSize);
197                 fileStats.endDownload(this, totalSize);
198             }
199         }
200     }
```

```

198             sawEnd = true;
199             return null;
200         }
201     } else {
202         DataBlock block = new DataBlock(size, buffer);
203         totalSize += size;
204         return block;
205     }
206 }
207 }
208 catch(Exception ex) {
209     close();
210     throw new MiddlewareException(ex);
211 }
212 }
213 }
```

Вот несколько примеров записей регистрационного журнала, относящихся к службе информационных потоков:

```

2004-12-21 19:17:43.320 INFO : jqpublic:
Streamer.getDataFile('/surface/tactical/sol/120/jpeg/1P138831013ETH2809P2845L
2M1.JPG)
2004-12-21 19:17:43.324 DEBUG: Begin download of file
/surface/tactical/sol/120/jpeg/1P138831013ETH2809P2845L2M1.JPG'
2004-12-21 19:17:44.584 DEBUG: Completed download of file
'/surface/tactical/sol/120/jpeg/1P138831013ETH2809P2845L2M1.JPG':
1876 bytes
```

На рис. 20.6 показан график, содержащий полезную информацию, которую можно извлечь из регистрационных раскопок. На графике показывается тенденция объема загружаемой с серверов информации (количество файлов и количество загруженных мегабайтов) за период, охватывающий несколько месяцев миссии. На более коротких отрезках времени график может показать всплески активности, когда один из марсоходов обнаруживает что-нибудь интересное.

Файлы и байты, загруженные службой потоков СИР

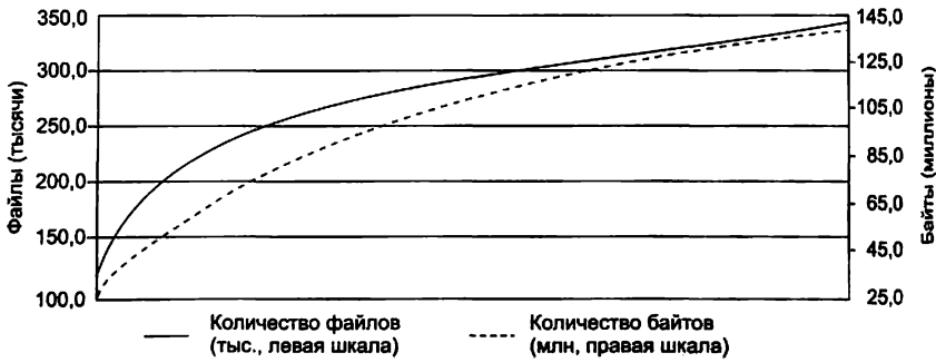


Рис. 20.6. График, созданный в результате «раскопок» регистрационного журнала службы потоков

Мониторинг

Регистрация позволяет нам анализировать производительность работы служб путем проверки всего, чем они занимались за какой-то период времени. В отличие от регистрационных записей, которые приносят наибольшую пользу в точном определении места возникновения проблем и причин их возникновения, мониторинг в процессе работы позволяет нам понять, насколько хорошо службы справляются со своей работой в данное время. Он дает нам возможность провести динамическую настройку с целью повышения производительности или воспрепятствования возникновению любых потенциальных проблем. Ранее уже упоминалось, что возможность отслеживания рабочего поведения зачастую является жизненно важной для успеха любого большого приложения.

В показанном ранее листинге есть операторы, которые обновляют данные о производительности, хранящиеся в глобальных статических объектах, на которые ссылаются поля `cacheStats` и `fileStats`. Служба мониторинга связующего уровня исследует эти данные по запросу. Глобальные объекты, на которые ссылаются эти поля, не показаны, но что они содержат, нетрудно представить. Главное, что сбор полезной информации о производительности работы не представляет особой сложности.

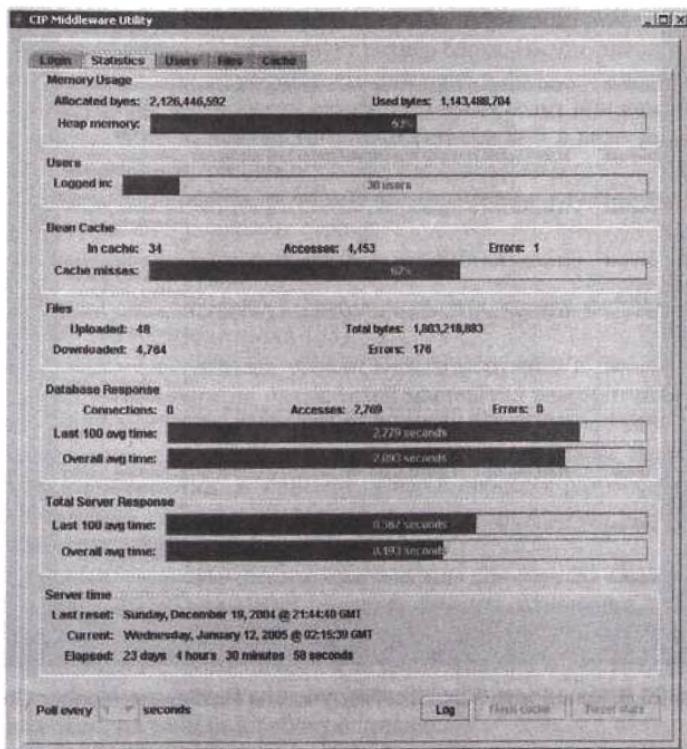


Рис. 20.7. Копия экрана вкладки Statistics утилиты Middleware Monitor Utility

Мы написали утилиту, осуществляющую мониторинг на связующем уровне CIP в виде клиентского приложения, которое периодически посылает запросы службе мониторинга связующего уровня для получения данных о текущей производительности. На рис. 20.7 показана копия экрана этой утилиты на вкладке Statistics (Статистика), на которой среди других рабочих статистических данных отображено количество файлов и байтов, которые были загружены и выложены службой потоков, и количество случившихся ошибок обработки файлов (среди которых неправильно указанные клиентским приложением имена файлов).

Оба метода, `doFileDownload()` и `readDataBlock()`, принадлежащие bean-компонентам, обслуживающим потоки, обновляют глобальный счетчик файловых ошибок (строки 50 и 84 в коде, показанном ранее в разделе «Ведение регистрационного журнала»). Методы `getDataFile()` и `readDataBlock()` увеличивают общее глобальное время отклика службы (строки 22 и 94). Как видно на рис. 20.7, утилита монитора связующего уровня отображает среднее время отклика во области с надписью Total Server Response.

Direction	Size	Start time	End time	Name
Download	144,545	Wed 2005.01.12 @ 01:02:02 GMT	Wed 2005.01.12 @ 01:02:06 OMT	/oss/merblops/ops/surf/acs/functional/sol/245/aps/actn/intsol_345_scren_capture.jpg
Download	145,861	Wed 2005.01.12 @ 00:56:38 GMT	Wed 2005.01.12 @ 00:56:42 OMT	/oss/merblops/ops/surf/acs/functional/sol/245/aps/actn/intsol_345_scren_capture.jpg
Download	32,950	Wed 2005.01.12 @ 00:35:44 OMT	Wed 2005.01.12 @ 00:35:45 OMT	/opt/bea/user_projects/cip/config/preferences/jdbcchain.preferences
Download	13,453	Wed 2005.01.12 @ 00:35:41 GMT	Wed 2005.01.12 @ 00:35:42 OMT	/opt/bea/user_projects/cip/config/global.properties
Download	23,629	Wed 2005.01.12 @ 00:25:55 OMT	Wed 2005.01.12 @ 00:25:57 OMT	/opt/bea/user_projects/cip/config/preferences/smcenna.preferences
Download	13,453	Wed 2005.01.12 @ 00:25:51 GMT	Wed 2005.01.12 @ 00:25:53 OMT	/opt/bea/user_projects/cip/config/global.properties
Download	14,482	Wed 2005.01.12 @ 00:16:26 GMT	Wed 2005.01.12 @ 00:16:26 OMT	/opt/bea/user_projects/cip/config/preferences/dwbatch.preferences
Download	13,453	Wed 2005.01.12 @ 00:16:25 GMT	Wed 2005.01.12 @ 00:16:25 OMT	/opt/bea/user_projects/cip/config/global.properties
Upload	22,016	Wed 2005.01.12 @ 00:13:04 OMT	Wed 2005.01.12 @ 00:13:05 OMT	/opt/bea/user_projects/cip/config/preferences/mile.preferences
Download	13,364	Tue 2005.01.11 @ 23:12:42 GMT	Tue 2005.01.11 @ 23:12:44 GMT	/opt/bea/user_projects/cip/config/preferences/cdrumble.preferences
Download	13,453	Tue 2005.01.11 @ 23:12:39 OMT	Tue 2005.01.11 @ 23:12:41 OMT	/opt/bea/user_projects/cip/config/global.properties

Poll every 1 seconds

Рис. 20.8. Копия экрана вкладки Files утилиты Middleware Monitor Utility

Метод `getDataFile()`, принадлежащий bean-компонентам чтения, фиксирует начало загрузки файла (строка 172). Метод `readDataBlock()` увеличивает пока-

зания глобальных итоговых счетчиков файлов и байтов (строки 195 и 196) и фиксирует завершение загрузки (строка 197). На рис. 20.8 показана копия экрана вкладки *Files* (Файлы) утилиты монитора, на которой отображены текущие и недавно осуществленные загрузки и выкладки файлов.

Запас прочности

Изменения неизбежны, и красивый код может изящно справляться с ними даже после запуска в эксплуатацию. Чтобы СИР обладал необходимым запасом прочности и мог справиться с возникающими изменениями, мы предприняли ряд мер:

- мы избежали жестко заданных параметров в связующих службах;
- мы обеспечили возможность вносить изменения в связующие службы, находящиеся в эксплуатации с минимальными задержками работы клиентских приложений.

Динамическая реконфигурация

У большинства связующих служб есть определенные ключевые параметры работы. Например, ранее мы видели, что служба потоков загружает содержимое файлов поблочно, а значит, существует размер блока. Вместо того чтобы жестко задать размер блока, мы поместили его значение в файл параметров, который считывается службой при каждом старте. Это происходит при каждой загрузке bean-компоненты, обслуживающего потоки (строки 3–5 в коде раздела «Ведение регистрационного журнала»).

Файл *middleware.properties*, который все связующие службы совместно используют и загружают, содержит строку:

```
middleware.streamer.blocksize = 65536
```

Затем метод *readDataBlock()* bean-компонента чтения файла может обратиться к этому значению (строка 183).

Каждая служба связующего уровня может загружать в начале работы несколько параметров. Один из показателей мастерства разработчика программного обеспечения заключается в знании, какие ключевые значения службы выставлять в качестве загружаемых параметров. Эти параметры, конечно, полезны и во время разработки. К примеру, у нас в процессе разработки была возможность испытать работу с различными размерами блоков, не проводя каждый раз перекомпиляции кода службы потоков.

Но загружаемые параметры еще более важны для запуска кода в эксплуатацию. Во многих промышленных областях вносить изменения в запущенное в эксплуатацию программное обеспечение – слишком сложное и дорогостоящее занятие. Конечно, то же самое было справедливо и к для миссии MER, которая имела официальный контролирующий орган по внесению изменений после того, как миссия вошла в рабочую стадию.

Отказ от жестко заданных значений параметров, конечно, является сто первой заповедью программирования, которая применима и маленьким, и к большим

приложениям. Но особую важность она приобретает для больших приложений, у которых может быть еще больше значений параметров, разбросанных по всему обширному пространству программного кода.

Горячая замена

Горячая замена является весьма важным свойством промышленного сервера приложений, который был задействован на связующем уровне CIP. Она давала возможность развернуть связующую службу, которая заменяла уже запущенную службу, не допуская предварительного сбоя в работе связующего уровня (и всего портала CIP).

Мы использовали горячую замену при необходимости заставить службу перегрузить значения ее параметров после изменений ситуации, достигая ее простой перезагрузкой службы на ее самом верхнем уровне. Разумеется, такая служба, как служба потоков, в которой используются bean-компоненты сессии, сохраняющие состояние (bean-компоненты чтения и записи файла), потеряют всю информацию о своем состоянии. Поэтому проводить горячую замену подобной службы мы могли только в периоды «затишья», когда знали, что в данный момент служба не задействована. Чтобы выбрать подходящий момент, мы могли воспользоваться вкладкой Files (Файлы) утилиты Middleware Monitor Utility (см. рис. 20.8).

Горячая замена имеет больше смысла, когда речь заходит о больших корпоративных приложениях, в которых важно сохранять работоспособность всего остального приложения, пока производится замена какой-то его составляющей. А небольшие приложения для внесения изменений проще всего, наверное, перезапустить.

Вывод

Совместный информационный портал – Collaborative Information Portal доказал, что даже в таких громадных правительственные агентствах, как NASA, возможно разрабатывать большие, сложные корпоративные системы программного обеспечения в сроки, позволяющие успешно соответствовать строгим требованиям функциональности, надежности и запаса прочности.

Марсоходы превзошли все ожидания, доказав, насколько хорошо аппаратное и программное обеспечение как для Марса, так и для Земли было спроектировано и построено и насколько высоко было мастерство его создателей.

В отличие от более мелких программ, красота больших приложений может быть найдена и вне изящных алгоритмов. Для CIP красота заключалась в реализации этой системы в виде архитектуры, предназначеннной для обслуживания широкого круга запросов, и в многочисленных простых, но хорошо подобранных компонентах – тех самых «гвоздиках», которые мастер конструирования программного обеспечения знает, куда загнать.

21 ERP5: Конструирование с целью достижения максимальной адаптивности

*Роджерио Эйтем де Карвальо (Rogerio Atem de Carvalho)
и Рафаэль Моннерат (Rafael Monnerat)*

Системы планирования ресурсов предприятия (Enterprise resource planning, ERP) известны, в основном, как большие, специализированные и очень дорогостоящие продукты. В 2001 году к работе ERP-системой с открытым кодом, известной как ERP5 (<http://www.erp5.com>), приступили две французские компании, *Nexedi* (в качестве основного разработчика) и *Coramty* (в качестве первого пользователя). Система ERP5 получила свое название в честь пяти основных понятий, составляющих ее ядро. Она основывается на проекте Zope и на языке написания сценариев Python, которые также являются системами с открытым кодом.

Мы поняли, что ERP5 очень легко поддается усовершенствованию и она легко настраивается как разработчиками, так и пользователями. Одной из причин стало то, что мы приняли передовой подход, в центре которого находилось понятие документа, а не процесса или данных. Основная идея, заложенная в понятии ориентации на документ, заключалась в том, что любой рабочий процесс зависит от подборки документов, позволяющих ему состояться. Поля документов соотносятся со структурой процесса, то есть поля отражают данные и возникающие между ними связи. Таким образом, если понаблюдать, как специалисты, использующие систему ERP5, осуществляют переходы по документам, можно раскрыть весь технологический процесс.

Технология, основанная на этом подходе, представлена инструментами и понятиями среды управления контентом – Content Management Framework (CMF), разработанной в рамках проекта Zope. Каждый экземпляр CMF, имеемый порталом, содержит объекты, в которых он размещает службы, среди которых просмотр, распечатка, документооборот и хранение. Структура документа реализована в виде класса портала на языке Python, и его поведение выполнено в виде портального делопроизводства.

Поэтому пользователи взаимодействуют с веб-документами, которые фактически являются представлениями системных объектов, управляемых специфическим документооборотом.

В этой главе будет показано, как эта документоориентированная парадигма и единый набор основных понятий делают ERP5 очень гибкой системой планирования ресурсов. Мы проиллюстрируем эти идеи путем объяснения, как мы воспользовались технологиями ускоренной разработки для создания модуля ERP5 под названием Project, который предназначен для управления проектом.

Общие цели ERP

ERP является программным продуктом, нацеленным на объединение всех данных и процессов, происходящих в организации, в единую систему. Поскольку это действительно непростая задача, производство ERP предлагает различные версии одного и того же программного обеспечения ERP для различных отраслей экономики, к примеру, нефтяной и газовой промышленности, машиностроения, фармацевтической промышленности, производства автомобилей, а также правительства.

Программное обеспечение ERP обычно состоит из ряда модулей, автоматизирующих работу организации. Наиболее распространенными модулями являются финансовый, управления запасами, зарплаты, планирования и управления производством, продажи и ведения бухгалтерского учета. Эти модули разработаны с учетом требований заказчика и их адаптации под пользователя, поскольку даже при том что организации, относящиеся к одной и той же отрасли экономики, придерживаются общего установленного порядка, каждая организация желает адаптировать ERP-систему под свои специфические нужды. К тому же программное обеспечение ERP быстро развивается, чтобы соответствовать развитию бизнеса, которому оно служит, и со временем в него добавляется все больше и больше модулей.

ERP5

ERP5 разрабатывается и используется растущим сообществом деловых и научных кругов Франции, Бразилии, Германии, Люксембурга, Польши, Сенегала, Индии и других стран. Система предлагает интегрированные решения по управлению бизнесом на основе платформы Zope, имеющей открытый код (<http://www.zope.org>), написанные на языке Python (<http://www.python.org>). К ключевым компонентам Zope, использованным в ERP5, относятся следующие.

ZODB

База данных объектов.

DCWorkflow

Движок документооборота.

Content Management Framework (CMF)

Инфраструктура, предназначенная для добавления и перемещения контента.

Zope Page Templates (ZPT)

Быстродействующая система создания сценариев на основе XML, имеющая графический пользовательский интерфейс (GUI).

Кроме того, ERP5 сильно зависит от XML-технологий. Каждый объект может быть экспортирован и импортирован в XML-формате, и два и более ERP5-сайтов могут совместно использовать синхронизированные объекты посредством SyncML-протокола. ERP5 также реализует схему объектно-реляционного отображения, которая хранит проиндексированные свойства каждого объекта в реляционной базе данных и позволяет вести поиск и извлечение объектов намного быстрее, чем ZODB. При этом способе объекты хранятся в ZODB, а поиск ведется с использованием SQL, являющимся стандартным языком запросов.

ERP5 был задуман как очень гибкая структура для разработки приложений для предприятий. Гибкость означает адаптивность к различным бизнес-моделям без особых затрат на изменения и поддержку. Чтобы достичь этого, необходимо определить основную объектно-ориентированную модель, от которой могут быть легко получены новые компоненты для определенных целей. Эта модель должна обладать достаточной абстрактностью, чтобы охватить все основные бизнес-понятия.

Судя по названию, в ERP5 определены пять абстрактных понятий, положенных в основу представления бизнес-процессов.

Resource (Ресурс)

Описывает ресурс, необходимый для осуществления бизнес-процесса, к примеру, индивидуальное мастерство, изделия, машины и т. д.

Node (Узел)

Бизнес-объект, получающий и отправляющий ресурсы. Он может иметь отношение к физическому объекту (например к промышленному предприятию) или к абстрактному (например к банковскому счету). Существуют и метаузлы, то есть узлы, содержащие другие узлы, например компании.

Path (Путь)

Описывает, как узел получает доступ к ресурсам, которые ему нужны от другого узла. К примеру, путь может быть торговой процедурой, которая определяет, как клиент получает продукт от поставщика.

Movement (Перемещение)

Описывает перемещение ресурсов между узлами на заданный момент и за заданный период времени. Например, одно такое перемещение может быть отгрузкой сырья со склада на фабрику. Движение – это реализация путей.

Item (Предмет)

Уникальный образец ресурса. К примеру, привод компакт-дисков – это ресурс для сборки компьютера, а привод компакт-дисков шифр-компоненты 23E982 – это предмет.

Наряду с некоторыми другими понятиями, такими как Order (Заказ) и Delivery (Поставка), они формируют унифицированную бизнес-модель ERP5 (Unified Business Model, UBM). Как будет показано в этой главе, новый бизнес-процесс можно реализовать за счет сочетания и расширения этих пяти понятий. Взаимоотношения между пятью основными понятиями показаны на рис. 21.1. Путь (Path) связан с узлом поставщика, который посыпает Ресурс (Resource) к узлу получателя. Перемещение (Movement) точно так же представляет перемещение предмета, который описан Ресурсом (Resource), от узла поставщика к узлу получателя.



Рис. 21.1. Основные классы ERP5

Базовая платформа Zope

Чтобы понять, почему про систему ERP5 сказано, что она документоуправляма, необходимо сначала понять, как работают Zope и его среда управления контентом – Content Management Framework (CMF). Zope изначально разрабатывалась как среда управления веб-контентом, предоставляющая ряд служб для управления жизненным циклом веб-документов. Со временем люди стали замечать, что она может быть также использована для реализации любого рода приложений, основанных на веб-технологиях.

Согласно направленности Zope на веб-контекст, ее CMF является средой, предназначенной для ускоренной разработки приложений, основанных на типах информационного наполнения. Она предоставляет ряд служб, связанных с этими типами, среди которых автоматизация документооборота, поиск, защита информации, проектирование и тестирование. От Zope CMF унаследовала доступ к ее объектной базе данных – ZODB (Zope Object Database), которая предоставляет функции транзакций и откатов.

CMF реализует структурную часть приложений через CMF Types, поддерживаемых службой `portal_types`, которая в свою очередь является своего рода инструментом системного реестра для распознанных типов данного портала. Видимая часть типа портала является документом, который его представляет. Для реализации своего поведения типы портала имеют связанные с ним действия, создающие документооборот, который в свою очередь реализует бизнес-процесс. Действия, проводимые с документом, изменяют его состояние и реализованы в виде сценария на языке Python, который осуществляет некую бизнес-логику; например вычисление общей стоимости заказа. Благодаря этой среде при разработке приложения на ERP5 нам приходится мыслить в терми-

иах документов, которые содержат данные бизнес-процесса и чей жизненный цикл поддерживается системой документооборота, который реализует поведение бизнес-процесса.

Чтобы получить преимущества от CMF-структуре, код ERP5 разбит на четырехуровневую архитектуру, которая реализует последовательную концепцию преобразований, имея на самом верхнем уровне конфигурационные задачи.

Первый уровень включает пять основных концептуальных классов. Они не имеют кода и состоят только из основы. Это сделано только для того, чтобы предоставить простую документацию:

```
class Movement:
```

```
    """
```

Перемещение определенного количества ресурсов заданной разновидности от поставщика к получателю.

```
    """
```

На втором уровне находится настоящая реализация основных классов на языке Python. Но здесь классы все еще носят абстрактный характер. Тем не менее в классах уже присутствует Zope, и они наследуются от XMLObject, что предопределяет возможность преобразования любого объекта в последовательную XML-форму для синхронизации или экспортации.

Свойства класса организованы в таблицы. Эти *таблицы свойств* являются реконфигурируемым набором атрибутов, который облегчает создание различных представлений объектов, потенциально управляемых различными наборами методов классов. Кроме того, эти представления позволяют системным администраторам проводить настройку безопасности очень гибким и сложным образом.

К примеру, таблица SimpleItem имеет атрибуты title, short_title и description. Системный администратор может так установить схему безопасности, что некоторые пользователи смогут только просматривать эти атрибуты, а другие смогут производить в них записи.

```
class Movement(XMLObject):
```

```
    """
```

Перемещение определенного количества ресурсов заданной разновидности от поставщика к получателю.

```
    """
```

определение имени типа

meta_type = 'ERP5 Movement'

определение имени CMF-типа

portal_type = 'Movement'

Добавление основной защитной конфигурации Zope

add_permission = Permissions.AddPortalContent

этот тип вносится в список как приемлемый тип контента

isPortalContent = 1

Этот тип является допустимым для средств ускоренной разработки

приложений ERP5 (Rapid Application Development facilities)

isRADContent = 1

используется для торговых и товарных операций

isMovement = 1

Объявляемая безопасность

сохранение основных сведений о безопасности базовых классов

```

security = ClassSecurityInfo()
# по умолчанию, допускает аутентификацию пользователей для просмотра
# объекта
security.declareObjectProtected(Permissions.AccessContentsInformation)

# Объявляемые свойства
property_sheets = ( PropertySheet.Base
    . PropertySheet.SimpleItem
    . PropertySheet.Amount
    . PropertySheet.Task
    . PropertySheet.Arrow
    . PropertySheet.Movement
    . PropertySheet.Price
)

```

Третий уровень содержит метаклассы Meta, которые являются реализующими (instantiable) классами. На этом уровне классы уже представляют определенные бизнес-объекты.

```
class DeliveryLine(Movement):
    """


```

Объект DeliveryLine позволяет строкам быть реализованными в поставках – Deliveries (упаковочный лист – packing list, заказ – order, счет-фактура – invoice и т.д.). Он может включать расценку (за страховку, за таможенное оформление, за оформление счетов, за заказы и т.д.)

```

    """


```

```

meta_type = 'ERP5 Delivery Line'
portal_type = 'Delivery Line'

# Объявляемые свойства
# необходимо переопределить свойство property_sheets.
# унаследованное от Movement
property_sheets = ( PropertySheet.Base
    . PropertySheet.XMLObject
    . PropertySheet.CategoryCore
    . PropertySheet.Amount
    . PropertySheet.Task
    . PropertySheet.Arrow
    . PropertySheet.Movement
    . PropertySheet.Price
    . PropertySheet.VariationRange
    . PropertySheet.ItemAggregation
    . PropertySheet.SortIndex
)

```

И наконец, четвертый уровень представлен классами Portal classes, основанными на CMF. Это уровень, на котором происходит конфигурация. К примеру, на рис. 21.2 показана основная часть вкладки Properties (Свойства). Эта копия экрана показывает в частности свойства Task Report Line. Этот тип является реализацией метатипа Delivery Line. Любопытно отметить, что в этой вкладке могут быть добавлены новые таблицы свойств, но для инструментов нашего проекта они не нужны.

Name	Value	Type
Title	Task Report Line	string
Description		text
Icon	organisation_icon.gif	string
Product meta type	ERP5 Delivery Line	string
Product factory method	addDeliveryLine	string
Add permission		string
Init Script		string

Рис. 21.2. Вкладка Properties (Свойства)

На рис. 21.3 показана вкладка Actions (Действия), в которой перечислены действия, связанные с типом Task Report Line. Действия реализуют определенные службы для этого типа. На рисунке можно увидеть службы View (Просмотр) и Print (Печать).

Г	Name	View
	Id	view
	Action	string:\${object_url}/TaskLine_view
	Icon	
	Condition	
	Permission	View
	Category	object_view
	Visible?	F
	Priority	1.0
Г	Name	Price
	Id	price_view
	Action	string:\${object_url}/DeliveryLine_viewPrice
	Icon	
	Condition	object/hasCellContent
	Permission	View
	Category	object_view
	Visible?	F
	Priority	2.0

Рис. 21.3. Вкладка Actions (Действия)

Четырехуровневая структура, представляющая системные классы, облегчает наращивание функциональных возможностей и свойств платформы. Это также позволяет проводить весьма распространенную в ERP5 практику: создавать

новые типы порталов без создания в системе новых классов. Все, что остается сделать программистам, — это изменить внешний вид, поскольку основные понятия, имеющиеся в ERP5, могут представить объекты специфических бизнес-областей.

Например, понятие *Movement* (Перемещение) может быть использовано для представления как изъятия денежных средств в финансовом модуле, так и перемещения материалов со склада на фабрику в товарном модуле. Для этого мы создаем один тип портала для представления изъятия денежных средств и другой — для представления перемещения материалов, в каждом из которых будут использованы бизнес-понятия, появляющиеся в графическом интерфейсе пользователя (GUI).

Помимо использования основных свойств CMF, ERP5 также реализует несколько дополнительных свойств для увеличения производительности программирования. Возможно, самым интересным является понятие «менеджеры отношений», которое относится к объектам, отвечающим за сохранность взаимоотношений между парами объектов. Программирование логики взаимосвязей в каждом бизнес-классе часто становится утомительным и подверженным ошибкам. К тому же традиционный код взаимоотношения расширяет реализацию (обратные указатели, уведомления об удалении и т. д.) многих бизнес-классов, и этот процесс труднее отследить, поддержать и синхронизировать, чем при опосредованных подходах.

В ERP5 портальный сервис под названием *Portal Categories* (портальные категории) записывает все отношения один-к-одному, один-ко-многим и многие-ко-многим между группами связанных объектов. Методы запросов, получатели и установщики, а также код взаимоотношений генерируются автоматически.

Этот сервис содержит объекты *основной категории*, которые соединяют сотрудничающие классы, чтобы выполнить данный бизнес-процесс. Для каждой основной категории ERP5 автоматически генерирует все необходимые получатели и установщики. К примеру, источник основной категории — ссылка на объекты типа *Node* (узел). Если в данной реализации ERP5 класс *Order* (Заказ) сконфигурирован на наличие этой основной категории, система автоматически включит все методы и ссылки, необходимые для перемещений от заказов к узлам, а если нужно, и в обратном направлении.

Понятия, используемые в ERP5 Project

Чтобы привести пример программирования модулей ERP5, мы посвятим большую часть остального материала главы исследованию ERP5 Project, гибкого инструмента управления проектом, который может быть использован различными способами.

Благодаря быстрым изменениям и конкуренции, царящей в мировой бизнес-среде, проекты являются обычной формой, используемой фирмами для

разработки инновационных продуктов и услуг. Поэтому к управлению проектами возникает интерес во всех отраслях промышленности.

Но что такое проект? Согласно сведениям, представленным в Википедии, проект это «временное усилие, предпринятое для создание уникального продукта или услуги» (<http://en.wikipedia.org/wiki/Project>, последний просмотр 13 апреля 2007 года).

Уникальность проектов усложняет управление ими, что иногда справедливо даже для небольших проектов. Следовательно, потребность в управлении проектом, которое определяется как «дисциплина, связанная с такой организацией и управлением ресурсами, которая позволяет этим ресурсам произвести всю работу, необходимую для завершения проекта в рамках определенных масштабов, качества, времени и затрат» (http://en.wikipedia.org/wiki/Project_management, последний просмотр 13 апреля 2007 года).

Поэтому при управлении проектом должен контролироваться ряд данных, связанных с такими ресурсами, как денежные средства, время и люди, с той целью, чтобы все шло по плану. В связи с этим возникает необходимость в информационных инструментах, облегчающих анализ большого объема данных.

Впервые ERP5 Project был использован как «внутренний» инструмент управления проектом, чтобы обеспечить проекты создания вариантов ERP5. Впоследствии он был переделан для поддержки совершенно других типов проектов. Более того, этот инструмент может управлять планированием выполнения заказа везде, где проектная точка зрения может помочь промышленному планированию и управлению. Иными словами, ERP5 Project может быть приспособлен к любой ситуации, о которой есть интерес думать в терминах проекта, составленного из серии задач и поставленного в рамки определенных ограничений.

ERP5 дает возможность разработчикам повторно использовать текущие модули при производстве других модулей в качестве абсолютно новых изделий. Следуя этому понятию, новый бизнес-шаблон (business template, BT) создан на базе одного из существующих шаблонов.

К тому времени, когда началась реализация ERP5 Project, в ERP5 уже был торговый модуль – Trade BT. Поэтому команда разработчиков решила взять за основу Project модуль Trade, представив планирующую часть проекта в виде повторного использования логики, разработанной для торговых операций. После завершения первой версии Project они могли улучшить Project BT, а затем использовать эти усовершенствования для переделки Trade BT, сделав его еще более гибким.

В построении Project интересующими нас частями Trade были классы Order и Delivery. Эти классы, являющиеся также частью UBM, служили контейнерами для объектов Order Line и Delivery Line, которые, в свою очередь, составляли перемещения – Movements, содержащие заказанные и доставленные единицы учета, как показано на рис. 21.4. На этом рисунке все подклассы самого нижнего уровня являются порталными типами (portal types). Поэтому в своей основе они имеют такую же структуру, что и их суперклассы, но у каждого портального типа имеется свой особый GUI и изменения в его документообороте, чтобы он работал в соответствии с логикой управления проектом.

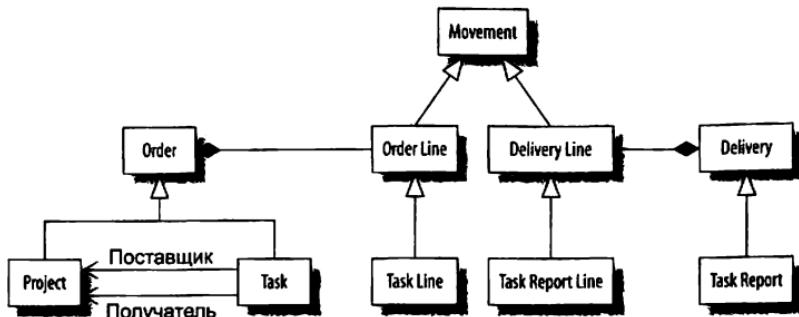


Рис. 21.4. Отношения между Trade и Project

Отношения между заказом *Order* и доставкой *Delivery* поддерживаются причинно-следственными связями, которые в основном определяют, что для каждого подтвержденного заказа будет когда-нибудь в будущем зеркально отраженная доставка. Задачи (*tasks*) и отчеты об их выполнении (*task reports*) наследуют такое же поведение. В соответствии с этим, строки заказа (*order lines*) представляют спланированные перемещения ресурсов между двумя узлами, которые после подтверждения будут выполнены и сгенерируют строки доставки (*delivery lines*). Следовательно, с точки зрения управления проектами, задачи обеспечивают планирование проекта, а отчеты об их выполнении — управление проектом.

Программирование ERP5 Project

Первое, о чем мы подумали при создании *Project BT*, был основной класс проекта. Поначалу вместо создания нового класса мы решили просто воспользоваться классом *Order* как таковым, без всяких изменений. Но по прошествии некоторого времени поняли, что бизнес-определения для заказа и проекта настолько различны, что нужно просто создать *Project* как подкласс *Order*, не добавляя никакого нового кода, только ради разделения понятий. В этом варианте конструкции *project* — это объект, который описывается последовательностью целей или этапов, с каждым из которых связана одна или несколько задач.

Затем нам нужно было решить, как реализовать управление задачами, учитывая разницу между проектными и торговыми операциями. Первое, что нужно было рассмотреть, — то, что задачи могут встречаться и вне рамок проекта — к примеру, в производственном планировании. Поэтому мы рассматривали задачу как композицию, состоящую из строк задачи, или небольших действий внутри задачи. Таким образом, мы отделяем задачи от проектов, позволяя им быть использованными в других ситуациях, сохраняя отношение с *Project* через основные категории исходного проекта — *source_project* и проекта назначения — *destination_project*.

Задачи реализуются через конфигурацию, как мы это делали для Task Report Line на рис. 21.1, с той лишь разницей, что Order используется как метакласс. Создание задачи, связанной с проектом, показано в следующем листинге:

```
# Добавление задачи в task_module. Context представляет текущий объект
# проекта.
context_obj = context.getObject()
# newContent - это ERP5 API метод, который создает новый контент.
task = context.task_module.newContent(portal_type = 'Task')
# Установка ссылки source_project на задачу.
task.setSourceProjectValue(context_obj)
# Перенаправление пользователя на Task GUI, чтобы он мог редактировать
# свойства задачи.
return context.REQUEST.RESPONSE.redirect(task.absolute_url() +
    '?portal_status_-
message=Created+Task.')
```

Запомните, что для извлечения задач конкретного проекта программисту нужно использовать только основную категорию source_project. С этой категорией ERP5 RAD автоматически генерирует характерные признаки и алгоритмы аксессоров. Интересно отметить, что те же самые аксессоры будут созданы как для Task, так и для Project. Используя конфигурацию, представленную вкладкой Actions (Действия), показанную на рис. 21.3, программист решает, какие из них использовать. В этой вкладке программист может определить новый GUI для использования следующих методов:

Эти аксессоры используются для перемещения от задачи к проекту

Этот метод возвращает связанную с Project ссылку
getSourceProject()

Этот метод устанавливает связанную с Project ссылку
setSourceProject()

Этот метод возвращает связанный с Project объект
getSourceProjectValue()

Этот метод устанавливает связанный с Project объект
setSourceProjectValue()

Эти аксессоры используются для перемещения от проекта к задаче

Этот метод возвращает ссылки на связанные задачи
getSourceProjectRelated()

Этот метод не сгенерирован, чтобы избежать нарушения инкапсуляции
setSourceProjectRelated()

Этот метод возвращает связанные объекты задач
getSourceProjectRelatedValue()

Этот метод не сгенерирован, чтобы избежать нарушения инкапсуляции
setSourceProjectRelatedValue()

Вы должны спросить, где же типичные, присущие проекту свойства и поведение. В большинстве случаев ответ по поводу свойств в том, что свойства

Movement и некоторых других UBM-классов замаскированы в GUI под другими именами. В других случаях свойство реализуется через основную категорию, со всеми аксессорами, которые генерируются согласно нашим ожиданиям.

Одним из примеров являются предшественники задач, то есть перечень задач, которые должны быть выполнены до выполнения данной задачи — вполне привычное понятие в управлении проектами, которое не встречается в торговых операциях. Этот перечень также реализуется с помощью основной категории под названием predecessor, которая связывает задачу с ее предшественниками конфигурируемым способом, поскольку категория берет на себя заботу обо всем необходимом коде.

Поведение задачи реализуется документооборотом. И тут снова повторно используется основное поведение Movement и более специфическое поведение Order. Эти документообороты управляют объектами тем способом, который имеет смысл для управления проектами, и включают некоторые способствующие этому сценарии. Документообороты облегчают разработку, поскольку они могут быть переконфигурированы, и программисту нужно написать сценарии только для специфической работы с объектами.

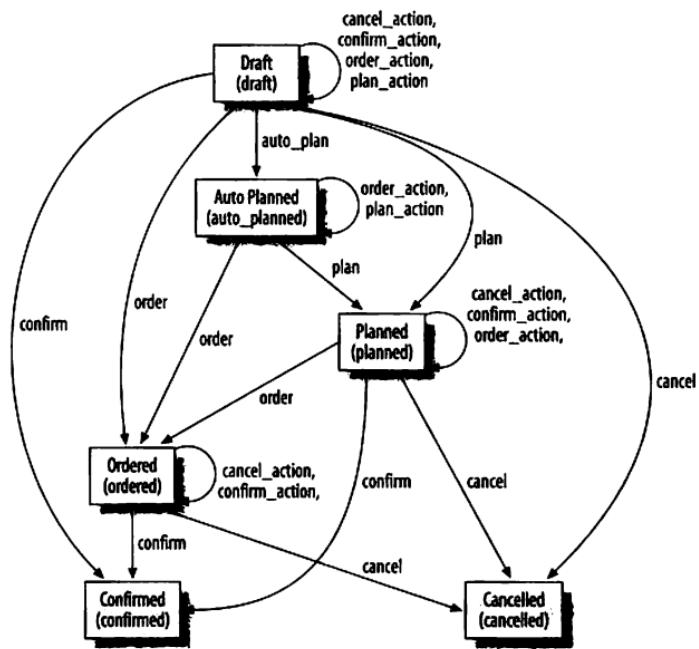


Рис. 21.5. Документооборот Task

На рис. 21.5 показан документооборот Task. В каждом прямоугольнике слова в скобках представляют структурный идентификатор. Перемещения с суффиксом _action переключаются благодаря событиям GUI; все остальные переключаются за счет событий документооборота. Для каждого перемещения можно

определить пред- и постсценарии. Эти сценарии и будут управлять объектами согласно бизнес-логике, в данном случае – логике выполнения задач. Задача представляет собой запланированный взгляд на процесс, который, по существу, проходит следующие стадии: запланировано (planned), предписано (ordered) и подтверждено (confirmed).

Этот документооборот точно такой же, как и в Order, но некоторые сценарии были изменены в соответствии с логикой, присущей проекту. В качестве примера далее приводится сценарий `order_validateData`, вызываемый перед каждым перемещением, имеющим суффикс `_action`:

```
## Этот сценарий проверяет наличие в Task необходимых данных
# получение используемого объекта задачи
task = state_change.object
error_message = ''
message_list = []
# проверка, подключена ли эта задача к какому-нибудь проекту или нет
if task.getSource() is None:
    message_list.append('No Source')
# если начальная дата равна нулю, но есть дата окончания, приравнять
# начальную дату к конечной: initialDate = finalDate
if task.getStartDate() is None and task.getStopDate() is not None:
    task.setStartDate(task.getStopDate())
if task.getStartDate () is None:
    message_list.append("No Date")
if task.getDestination() is None:
    message_list.append('No Destination')
# каждый содержащийся объект фильтруется на принадлежность к
# перемещениям.
# Типичными возвращаемыми значениями будет что-нибудь похожее на
#('Task Line', 'Sale Order Line', 'Purchase Order Line')
for line in
    task.objectValues(portal_type=task.getPortalOrderMovementTypeList()):
        # проверка, имеют ли все перемещения связанные с ними ресурсы
        if line.getResourceValue() is None:
            message_list.append("No Resource for line with id: %s" % line.getId())
# если произойдет какая-либо ошибка, инициализировать предупреждение
if len(message_list) > 0:
    raise ValidationFailed, "Warning: " + " --- ".join(message_list)
```

На рис. 21.6 показан документооборот Task Report. Он следует той же логике, что и документооборот Delivery, и использует некоторые дополнительные сценарии, такие как показанный здесь `taskReport_notifyAssignee`.

```
task_report = state_change.object
# поиск ответственного за выполнение задачи
source_person = task_report.getSourceValue(portal_type="Person")
# поиск ответственных
destination_person = task_report.getDestinationValue(portal_type="Person")
# получение электронного почтового адреса ответственного
if source_person is not None:
    from_email = destination_person.getDefaultEmailText()
    email = source_person.getDefaultEmailValue()
    if email is not None:
        msg = """
# сюда помещается заранее сформированная строка с сообщением и данными
```

задачи

```
"""
email.activate().send(from_url = from_email,
                      subject="New Task Assigned to You",
                      msg = msg)
```

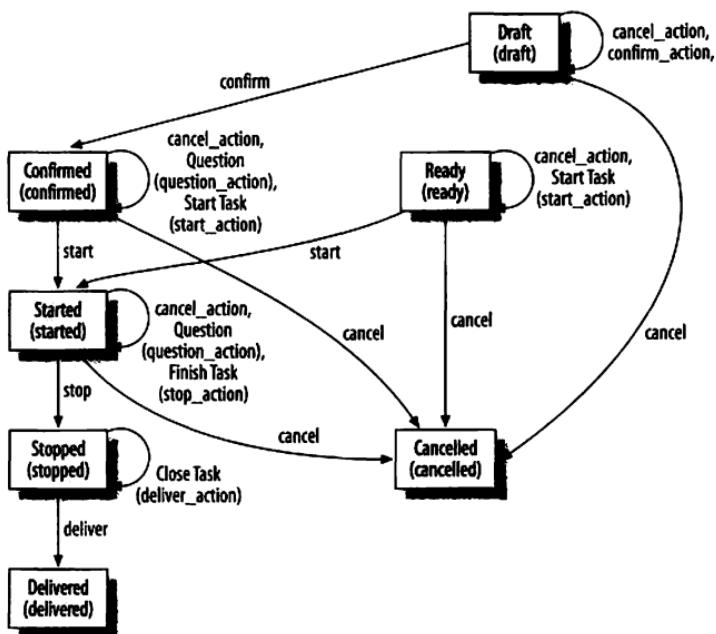


Рис. 21.6. Документооборот Task Report

Вывод

Команде ERP5 удалось реализовать очень гибкий инструмент, используемый как для «традиционного» управления проектами, так и для планирования заказов и контроля выполнения за счет существенного повторного использования уже существующих основных понятий и кода. По сути, повторное использование — это повседневная практика ERP5-разработки, а совершенно новые модули создаются лишь для того, чтобы изменять элементы GUI и регулировать документооборот.

Благодаря этому акценту на повторное использование запросы к базе данных объектов могут быть сделаны на абстрактном уровне портальных типов или метаклассов. В первом случае извлекаются специфические, присущие данному делу бизнес-концепции, например задачи проекта. А во втором — извлекаются все объекты, связанные с универсальными понятиями UBM, что весьма интересно для таких потребностей, как сбор статистики.

В этой главе мы подвергли редактированию некоторые фрагменты кода, чтобы улучшить их читаемость. Весь код ERP5 в своем необработанном виде доступен по адресу: <http://svn.erp5.org/erp5/trunk>.

Благодарности

Мы хотели бы поблагодарить Жан-Поля Сметса-Соланеса (Jean-Paul Smets-Solanes), создателя и главного разработчика ERP5, и всех представителей его команды, особенно Романа Корто (Romain Courteaud) и Тьери Фаше (Thierry Faucher). Когда во время обсуждения конструкции и реализации ERP5 авторы говорят «мы», они относят это понятие ко всем замечательным ребятам из *Nexedi*.

22 Ложка грязи

Брайан Кэнтрилл (Bryan Cantrill)

Если в бочку вина добавить ложку грязи, то получится бочка грязи.

Закон энтропии Шопенгауэра

В отличие от всех других создаваемых нами конструкций, годность программного обеспечения имеет двоичное представление: программа либо работает, либо не работает. То есть в отличие от моста, самолета или микропроцессора, у программы нет физических параметров, ограничивающих рамки ее пригодности; программа не обладает расчетной нагрузкой, максимальной скоростью или климатическими условиями эксплуатации. В этом отношении она больше напоминает математическое доказательство, чем физический механизм: изящность или грубоść доказательства — понятие субъективное, но его правильность — нет.

И это действительно придает программному обеспечению ту чистоту, которая традиционно присуща лишь математике: подобно ей программа может быть правильной в абсолютном и вечном смысле. И если эта чистота программы является ее доктором Джекиллом, то присущая ей хрупкость является ее мистером Хайдом: учитывая, что программа может быть либо правильной, либо неправильной, один-единственный дефект может составить разницу между несомненным успехом и полным провалом.

Разумеется, это не говорит о том, что каждый дефект непременно будет fatalным, но всегда существует возможность, что единственный дефект выльется в нечто более существенное, чем его обнаружение: в конструктивный недостаток, подвергающий сомнению фундаментальные положения, на которых строилась вся программа. Такой дефект может потрясти программу до основания и, в худшем случае, полностью сделать ее непригодной. То есть один-единственный программный дефект может стать той самой пресловутой ложкой грязи в бочке вина, превращая то, что должно было радовать, в сплошную отраву.

Лично для меня рубеж, отделяющий вино от грязи, никогда не был настолько явным, как в одном из случаев, связанных с разработкой важной подсистемы ядра операционной системы Solaris в 1999 году. Возникшая проблема и ее решение заслуживают пристального рассмотрения, поскольку показывают нам, как глубокие конструктивные дефекты могут проявиться в виде ошибок и какими коварными могут стать отдельные компоненты, когда они затрагивают сложную и жизненно важную основу нормального функционирования программы.

Перед началом следует предупредить: это путешествие уведет нас в глубины ядра Solaris, в недра наиболее существенного и тонкого механизма операционной системы. Подробности могут показаться слишком тягостными; мы, как рисующие жизнью спелеологи, временами будем пробираться впопыхах, погружаясь в холодную воду, или протискиваться в узкие, не дающие свободно дышать проходы, но тех, кто пустился в это путешествие, ждет скрытая от посторонних глаз прекрасная подземная пещера. Итак, если вы готовы, прикрепите наплечный фонарь, прихватите фляжку с водой и спускайтесь в ядро Solaris...

В центре всего повествования будет фигурировать подсистема турникетов. *Турникет* является механизмом, используемым для блокировки и активизации потоков в Solaris – это усиление базисных элементов (примитивов) синхронизации, таких как объекты-мьютексы и блокировки чтения-записи. Пускай программный код скажет все сам за себя¹:

```
/*
* Турникеты обеспечивают блокировку и поддержку активизации, включая
* приоритетное наследование, для примитивов синхронизации
* (например мьютексов и блокировок чтения-записи – rwlocks).
* Их обычное использование выглядит следующим образом:
*
* Для блокировки 'lp' на доступ к чтению в foo_enter():
*
*     ts = turnstile_lookup(lp);
*     [ Если блокировка все еще удерживается, установка бита ожидания
*       turnstile_block(ts, TS_READER_Q, lp, &foo_sobj_ops);
*
* Для активизации потоков, ожидающих доступа по записи к заблокированному
* 'lp' в foo_exit():
*
*     ts = turnstile_lookup(lp);
*     [ Либо сброс блокировки (изменение владельца на NULL) либо выполнение
*       [ непосредственной передачи (смена владельца одного из потоков, который
*         [ мы собираемся активизировать).
*     [ Если мы собираемся активизировать последнего ожидающего, сброс бита
*       [ ожидания.
*     turnstile_wakeup(ts, TS_WRITER_Q, nwaiters, new_owner or NULL);
*
* turnstile_lookup() возвращает хранящийся в кэш турникета с блокирующей цепочкой
* для lp.
* Обе функции, и turnstile_block() и turnstile_wakeup(), сбрасывают турникетную
```

¹ Это открытый код, доступный по адресу: <http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/os/turnstile.c>.

- * блокировку.
- * Для отмены турникетной операции клиент должен вызвать `turnstile_exit()`.
- *

Таким образом, турникетная абстракция позволяет примитивам синхронизации сосредоточиваться на своем собственном специфическом образе действий, не заботясь о тонкостях механизмов блокировки и активизации. Как упоминалось в комментарии к программному блоку, `turnstile_block()` — это функция, вызываемая для фактической блокировки текущего потока примитивом синхронизации, и именно с этой функции, с моего загадочного комментария, и начинается наше настоящее подземное путешествие.

```
/*
 * Перемещение по блокирующей цепочке к ее окончанию. завещая наш приоритет
 * всем, кто встречается на пути.
 */
while (t->t_sobj_ops != NULL &&
       (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
    if (owner == curthread) {
        if (SOBJ_TYPE(sobj_ops) != SOBJ_USER_PI) {
            panic("Взаимная блокировка: цикл в блокирующей цепочке");
        }
    /*
     * Если обнаруженный цикл заканчивается в mp,
     * то мы знаем, что это не 'настоящий' цикл, поскольку
     * мы собираемся сбросить mp перед тем как дезактивироваться.
     * Кроме этого, после того как мы пройдем полный цикл,
     * мы знаем, что должны завещать приоритет всем на своем пути.
     * Поэтому теперь мы можем его прервать.
     */
    if (t->t_wchan == (void *)mp)
        break;
```

Для меня этот комментарий и две строки кода, на которые он ссылается (которые выделены полужирным шрифтом), будут всегда служить каноническим примером той самой разницы между грязью и вином: они были добавлены в завершающий, самый суматошный период разработки Solaris 8, в один из самых напряженных моментов моей инженерной карьеры — в период недельной совместной работы с коллегой Джефом Бонвиком (Jeff Bonwick), инженером компании *Sun*. Эта работа потребовала такой общности душевного состояния, что мы оба придумали ей название «сплав разума».

Мы еще вернемся к этому коду и к стоящему за ним объединенному разуму, но чтобы до всего этого добраться, сначала нужно углубиться во внутреннюю работу турникетов, исследовав, в частности, как турникеты обращаются к классической проблеме *инверсии приоритетов*.

Если вы не знакомы с этой проблемой, ее можно описать следующим образом: если есть три потока с тремя различными приоритетами и если поток, имеющий высший приоритет, блокируется объектом синхронизации, удерживаемым потоком с самым низким приоритетом, поток со средним приоритетом может (в простой вытесняющей системе приоритетов, запущенной на однопроцессорной системе) работать бесконечно, подавляя поток с более высоким приоритетом. Результат показан на рис. 22.1.



Рис. 22.1. Инверсия приоритетов

Один из механизмов решения проблемы инверсии приоритетов представляет собой технологию под названием *наследование приоритетов*. При наследовании приоритетов, когда один поток собирается заблокироваться на ресурсе, удерживаемом потоком с более низким приоритетом, поток с более высоким приоритетом *завещает* свой приоритет потоку с более низким приоритетом на время продолжительности критического участка программы. То есть приоритет потока с более низким приоритетом *повышается* до того уровня, которым обладает поток с более высоким приоритетом, до тех пор пока поток с более низким приоритетом будет являться владельцем ресурса, необходимого потоку с более высоким приоритетом. Когда поток с более низким приоритетом (работающий с повышенным приоритетом) выйдет из критического участка программы — когда он освободит примитив синхронизации, на котором заблокировался поток с более высоким уровнем синхронизации, — он активизирует поток с более высоким приоритетом и возвращает себя на более низкий приоритет. Таким образом, никакой поток со средним уровнем приоритета никогда не сможет запуститься, и инверсия будет предотвращена.

Теперь в Solaris у нас есть достаточно продолжительный опыт использования наследования приоритетов для примитивов синхронизации ядра; несомненно, в этом и есть одно из архитектурных отличий SunOS 4.x и Solaris 2.x, которое заключается в одной из основных служб подсистемы турникетов. Но получение наследования приоритетов именно для примитивов синхронизации ядра довольно опасно: нужно знать, кто является владельцем блокировки, и нужно знать, чьей блокировкой заблокирован сам владелец (если такое имеет место). То есть если поток заблокирован блокировкой, которой владеет поток, который *тоже* заблокирован, у нас должна быть возможность определить, какая блокировка заблокировала поток-владелец и какой поток владеет *этой* блокировкой. Мы называем эту цепочку заблокированных потоков *блокирующей цепочкой*, и поскольку ее детали являются самыми важными в реализации

наследования приоритетов, есть смысл досконально разобрать вполне правдоподобный и конкретный пример возможной разработки.

В качестве примера блокирующей цепочки мы можем рассмотреть взаимодействие двух хорошо известных подсистем Solaris: распределителя памяти ядра и файловой системы Zettabyte Filesystem (ZFS). Исходя из целей нашего примера, нам не нужно разбираться во всех тонкостях этих довольно объемных подсистем; мы выясним в них лишь нужные нам места, не исследуя многочисленные щели и закоулки. Характерная особенность распределителя памяти ядра состоит в том, что он является *объектно-кэширующим* (*object-caching*) распределителем — все назначения обслуживаются из кэшей, которые работают с объектами фиксированного размера, — и поэтому его кэши, распределяющие буферы по структурам, рассчитанным на каждый CPU, называются *хранилищами* (*magazines*). Когда хранилище истощается, распределения удовлетворяются из структуры под названием *склад* (*depot*). Эта многоярусная структура хорошо масштабируется относительно центральных процессоров (CPU): удовлетворяя большинство распределений из хранилища, имеющего структуру для каждого CPU (при которой конкуренция маловероятна), распределитель проявляет почти линейную масштабируемость по CPU. И хотя это не отвечает нашим сиюминутным целям, я не могу удержаться от рассмотрения одной изящной тонкости в коде, который выполняется, когда выделенное CPU хранилище исчерпано и должна быть захвачена блокировка склада (которая носит для каждого кэша глобальный характер):

```
/*
 * Если мы не можем получить блокировку depot без конфликтов,
 * нужно обновить счетчик конфликтов. Мы используем уровень
 * конфликтов, связанных с depot, чтобы определить необходимость увеличения
 * размеров хранилища (magazine) для достижения лучшей масштабируемости.
 */
if (!mutex_tryenter(&cp->cache_depot_lock)) {
    mutex_enter(&cp->cache_depot_lock);
    cp->cache_depot_contention++;
}
```

Этот код не просто захватывает блокировку, он предпринимает попытки захвата блокировки, отслеживает количество попыток, неудавшихся из-за ее удержания. Получающееся количество является приблизительным показателем конфликтов на глобальном уровне, и если количество становится слишком большим за определенный период времени, система увеличивает количество буферов, запасенных на уровне каждого CPU, сокращая количество конфликтов на глобальном уровне. Таким образом, этот незамысловатый механизм позволяет подсистеме динамически подстраивать ее структуру для сокращения количества своих собственных конфликтов! Код, несомненно, красивый.

Но вернемся к объяснению нашего примера и теперь обратимся к ZFS, о которой нам нужно знать лишь то, что файлы и каталоги обладают хранящейся в памяти структурой, названной *znode*.

С учетом этих исходных сведений о распределителе памяти ядра и ZFS мы можем предположить следующую последовательность событий.

1. Поток T1 предпринимает попытку получить распределение памяти от кэша `kmem_alloc_32` на CPU 2, что требует блокировки принадлежащего CPU 2 хранилища `kmem_alloc_32`. Обнаружив, что все хранилища для CPU 2 исчерпаны, T1 захватывает блокировку склада (`depot`) для `kmem_alloc_32 cache` и тогда приобретает преимущественное право как на удержание блокировки хранилища CPU 2, так и на удержание блокировки склада.
2. Второй поток, T2, запущенный на CPU 3, пытается получить не связанное с этим распределение памяти от кэша `kmem_alloc_32`. По неблагоприятному стечению обстоятельств его хранилища также исчерпаны. T2 пытается захватить блокировку склада для кэша `kmem_alloc_32`, но видит, что блокировка удерживается потоком T1, вследствие чего он блокируется.
3. Третий поток, T3, запускается на CPU 3 после того, как T2 уже заблокирован. Этот поток пытается создать ZFS-файл `/foo/bar/mumble`. Как часть этой операции, он должен создать блокировку записи в ZFS-каталог для записи `mumble` в каталог `/foo/bar`. Он захватывает блокировку структуры `znode`, которая соответствует `/foo/bar`, а затем предпринимает попытку распределить `zfs_dirlock_t`. Поскольку размер `zfs_dirlock_t` составляет 32 байта, это распределение должно быть удовлетворено из кэша `kmem_alloc_32`, и поэтому T3 пытается захватить блокировку хранилища для кэша `kmem_alloc_32` на CPU 3, но он видит, что эта блокировка удерживается потоком T2, и поэтому он блокируется.
4. Четвертый поток, T4, предпринимает попытку просмотреть содержимое каталога `/foo/bar`. В качестве части этой операции он пытается захватить блокировку структуры `znode`, которая соответствует `/foo/bar`, но видит, что блокировка удерживается потоком T3, поэтому он блокируется.

Когда T4 блокируется, он блокируется на потоке T3, который в свою очередь блокируется на потоке T2, а тот, в свою очередь, блокируется на потоке T1 — и именно эта цепочка потоков и составляет блокирующую цепочку. Видя, какой нелепой может оказаться блокирующая цепочка, может быть, проще понять жизненную необходимость искусственного получения правильного наследования приоритетов: когда заблокированный поток завещает свой приоритет своей блокирующей цепочке, мы должны осуществить последовательный перебор блокирующей цепочки *когерентно*. То есть когда мы это делаем, то должны видеть последовательную картину состояния всех потоков блокирующей цепочки на данный момент времени — не больше и не меньше. В контексте данного примера мы не хотим завещать наш приоритет потоку T1 *после* того, как он снимет блокировку, удерживающую поток T2 (и таким образом, по переходящей, поток T4) — это потенциально оставляло бы потоку T1 искусственно завышенный приоритет.

Но как осуществить последовательный перебор блокирующей цепочки, сблюдая при этом когерентность? В операционной системе Solaris состояние диспетчера потоков (например его работа, ожидание очереди на активизацию или бездействие) защищено благодаря захвату специальной спин-блокировки, известной как ее *блокировка потока*; возможно было бы заманчиво поверить в то, что стремясь к когерентности последовательного перебора блокирующей

цепочки, достаточно было бы всего лишь захватить сразу все блокировки потоков. Но эта схема неработоспособна, в частности, из-за самой природы осуществления блокировок потоков.

Блокировка потока является весьма специфичной блокировкой, поскольку в традиционном смысле она не является спин-блокировкой, скорее всего, она является *указателем* на спин-блокировку, а блокировка, на которую она указывает, является той самой блокировкой, которая защищает структуру, управляющую потоком в данный момент; как только управление потоком перейдет от одной структуры к другой, блокировка потока операционной системы переместит указатель на соответствующую блокировку.

Например, если поток поставлен в очередь на запуск на CPU, блокировка потока указывает на блокировку регулируемой очереди на этом CPU, на блокировку уровней приоритета потоков, но когда поток запущен на CPU, блокировка потока изменяется и указывает на блокировку структуры `sri_t` на этом CPU. А когда поток должен блокироваться на примитиве синхронизации, блокировка потока изменяется и указывает на блокировку (звучит тревожная музыка) в таблице турникета.

Последняя структура будет приобретать все большую важность по мере того, как мы будем спускаться в глубины нашей пещеры, но на данный момент для нас важно следующее: мы не в состоянии запросто захватить каждую блокировку потока, поскольку несколько потоков могут указывать на *одну и ту же* основную блокировку диспетчера; если бы мы просто попытались всех их захватить, мы могли бы попасть в ситуацию собственной взаимной блокировки в процессе осуществления попыток захвата уже захваченной нами блокировки¹¹.

К счастью, на самом деле нам не нужно удерживать каждую блокировку потока, чтобы обеспечить последовательную картину блокирующей цепочки, благодаря важному (если не очевидному) свойству блокирующих цепочек: их можно распутать только с *незаблокированных* концов. То есть единственным способом для потока, заблокированного примитивом синхронизации, стать разблокированным – это быть явным образом активизированным тем потоком, который является владельцем этого примитива.

В нашем примере единственный способ, скажем, для потока T3 стать работоспособным – это быть активизированным потоком T2. Таким образом, если мы продолжаем двигаться атомарно от T3 к T2, а затем атомарно от T2 к T1, мы получаем гарантию отсутствия какого-либо промежутка времени, в который может быть активизирован T3 – даже если мы уже сбросили блокировку потока для T3.

Это означает, что нам нужно блокировать *не всю* цепочку, а лишь *два ее последовательных элемента*: когда должен блокироваться T4, мы можем захватить блокировку для T3, затем захватить блокировку для T2, затем сбросить блокировку для T3 и овладеть блокировкой для T1, затем сбросить блокировку для T2 и т. д. Поскольку в каждый момент времени мы заботимся лишь об

¹¹ И тут возникает еще одна довольно острые проблема упорядочения блокировки; достаточно сказать, что захват всех блокировок потоков в блокирующей цепочке обречен на провал по множеству причин.

удержании двух блокировок потока, то становится довольно просто разобраться с тем случаем, когда они указывают на одну и ту же основную блокировку: тогда мы просто сохраняем эту блокировку по мере продвижения по элементам блокирующей цепочки.

Тем самым проблема перебора блокирующей цепочки *почти* решена, но существенные препятствия, в частности последствия разных конструкторских решений, все же остаются. Вернемся к уже упомянутому положению о том, что блокировки потоков могут указывать на блокировки диспетчера в таблице турникетов. Теперь нужно объяснить, что такое таблица турникетов, поскольку в нашем путешествии мы еще несколько раз с ней столкнемся.

Таблица турникетов — это хэш-таблица, ключи которой представляют собой виртуальные адреса примитивов синхронизации. Это таблица очередности, в которой выстроены заблокированные потоки. Каждая очередь заблокирована на своей верхушке *блокировкой турникета*, и это одна из тех блокировок, на которую укажет относящаяся к потоку блокировка потоков, если этот поток заблокирован примитивом синхронизации.

Это весьма рискованное, если не сказать коварное, конструкторское решение: когда поток заблокирован примитивом синхронизации, он *не* становится в ту очередь, которая является уникальной для этого элемента, а попадает в ту очередь, которая, возможно, совместно используется несколькими примитивами синхронизации, что приводит к отображению на одну и ту же запись в таблице турникета.

Почему так было сделано? Будучи операционной системой с высокой степенью параллельной обработки, Solaris обладает мелкомодульной синхронизацией, что означает наличие многочисленных (!) экземпляров примитивов синхронизации и очень частое и практически бесконфликтное управление ими. Соответственно структуры, представляющие примитивы синхронизации ядра — `kmutex_t` и `krlwlock_t` — должны быть как можно меньше, и управление ими должно быть оптимизировано для общего бесконфликтного применения. Внедрение очереди для блокирующей цепочки в сам примитив синхронизации приведет к недопустимому влиянию либо на пространство памяти (из-за разрастания размера примитива за счет указателя очереди и блокировки диспетчера), либо на время работы (за счет уменьшения бесконфликтности при управлении более сложной структурой). В любом случае недопустимо отводить место для структуры данных блокирующей цепочки в самом примитиве синхронизации, для этого требуется таблица турникета (или что-либо сходное с ней).

Возвращаемся к варианту таблицы турникета: у потоков, заблокированных *разными* примитивами синхронизации, могут быть блокировки потоков, указывающие на *одну и ту же* блокировку турникета. Учитывая, что при проходе по блокирующей цепочке мы должны удерживать в любой момент времени две блокировки, это создает угрожающую проблему упорядочения блокировок. Когда Джейф в свой исходной реализации столкнулся с этой проблемой, он нашел весьма изящный выход; его комментарий в `turnstile_interlock()` объясняет суть проблемы и принятное им решение.

```
/*
 * Применяя наследование приоритета, мы должны захватить блокировку
 * потока-владельца, одновременно удерживая блокировку ожидающего потока. Если
 * обе блокировки потоков являются блокировками турникета, это может привести
 * к взаимной блокировке: при удержании L1 и попытке захвата L2 какой-нибудь
 * не связанный с этим поток может применить наследование приоритета к какой-нибудь
 * другой блокирующей цепочке, удерживая L2 и пытаясь захватить L1. Наиболее
 * очевидное решение - применение lock_try() для владельца блокировки - является
 * недостаточным, поскольку может завестись в тупик: каждый поток может удерживать
 * по одной блокировке, которую пытается захватить другой поток, потерпеть при этом
 * неудачу, отступить и повторять попытку, попадая в бесконечный цикл.
 * Чтобы предупредить тупиковую ситуацию, мы должны определить
 * победителя, то есть определить произвольный порядок блокировки для блокировок
 * турникетов. Для простоты мы объявляем, что порядок виртуальной адресации
 * определяет порядок блокировки, то есть если L1 < L2, то соответствующий порядок
 * блокировки - это L1, L2. Таким образом поток, удерживающий L1 и желающий
 * захватить L2, должен стать в цикл ожидания, пока L2 не станет доступна, а поток,
 * который удерживает L2 и не может с первой попытки захватить L1, должен сбросить
 * L2 и вернуть сбой.
 * Кроме того, проигравший поток не должен заново захватывать L2 до тех пор,
 * пока выигравший поток имеет шанс его захватить; чтобы гарантировать это,
 * проигравший поток должен захватить L1 после сброса L2, став, таким образом,
 * в цикл ожидания, пока победивший не справится со своей задачей.
 * В дальнейшем, усложняя вопрос, следует заметить, что относящийся к владельцу
 * указатель на блокировку потока может измениться (то есть указывать на другую
 * блокировку), пока мы пытаемся его захватить. Если это произойдет, мы должны
 * распутать служившую ситуацию и повторить попытку.
 */
```

Эта проблема упорядочения блокировок является лишь частью того, что усложнило реализацию наследования приоритета для объектов синхронизации ядра, и, к сожалению, наследование приоритета на уровне ядра решает только часть проблемы инверсии приоритетов.

Предоставление наследования приоритетов исключительно для объектов синхронизации ядра имеет очевидный недостаток: для построения многопоточной системы реального времени необходимо иметь наследование приоритетов не только на уровне примитивов синхронизации ядра, но также и на *пользовательском уровне* примитивов синхронизации. И именно к этой проблеме – наследованию приоритетов на пользовательском уровне – мы решили обратиться в Solaris 8. Мы назначили специалиста для ее решения, и (под чутким руководством тех из нас, кто лучше других разбирался в тонкостях составления рабочего графика и синхронизации) в октябре 1999 года работа над новым свойством была завершена.

Спустя несколько месяцев, в декабре 1999 года, я занимался поиском причин сбоя операционной системы, обнаруженного моим коллегой. Сразу стало ясно, что некий дефект закрался в нашу реализацию наследования приоритетов на пользовательском уровне, но как только я осознал наличие ошибки, то сразу сообразил, что проблема лежит отнюдь не на поверхности: это был конструктивный просчет, и я практически мог по запаху определить, что наше вино превратилось в грязь.

Перед объяснением этой ошибки и вскрытием благодаря ей конструктивного просчета стоит рассмотреть методику, использованную при отладке. Важной

составляющей мастерства разработчика программного обеспечения является способность проводить анализ отказа сложной программной системы и представлять этот анализ во всех деталях. И как в любой достаточно сложной системе, анализ отказа должен быть аргументирован: он будет основан на картине состояния системы в момент отказа. Несомненно, подобная картина состояния настолько важна для отладки, что со временем начала компьютерной эпохи она носит прозвище «дамп ядра».

Этот вариант — «посмертная» отладка — может быть противопоставлен более традиционному варианту — отладке *на месте*, согласно которому отладка ведется на «живой» и работающей (хотя и остановленной) системе. Тогда как при отладке *на месте* для многократного тестирования гипотез относительно системы можно использовать контрольные точки, при «посмертной» отладке для проверки гипотез можно использовать лишь состояние системы на момент отказа. Хотя это означает, что при «посмертной» отладке требуется менее сложная методика, чем при отладке *на месте* (поскольку существуют такие дефекты, которые просто не проявляются в момент отказа, что существенно сокращает количество гипотез), но есть и немало дефектов, которые не воспроизводятся в достаточной мере для того, чтобы применить к ним отладку *на месте*, и для их устранения не остается ничего иного, кроме «посмертной» отладки.

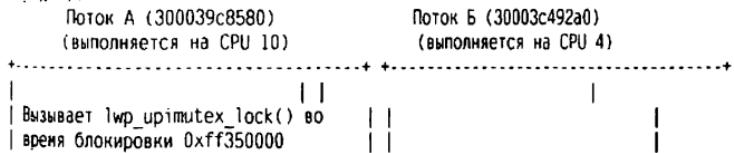
Кроме того, поскольку «посмертная» отладка предоставляет меньше возможностей, она требует более глубоких раздумий, как при построении гипотез, так и при их проверке, поэтому для развития способностей к отладке «посмертная» отладка подходит намного больше, чем отладка *на месте*.

И наконец, поскольку состояние системы статично, специалист по отладке может предоставить собственноручно составленный подробный анализ коллегам, которые затем параллельно с ним могут составить свой собственный анализ и сделать свои выводы. И даже если этот анализ не будет активно проверяться другими специалистами, он все равно будет полезен, его составление заставляет специалиста по отладке обратиться к пробелам в собственной логике. Короче говоря, «посмертная» отладка — важнейшая часть нашего ремесла — мастерство, которое должен развивать у себя каждый серьезный разработчик программного обеспечения.

После изложения основ (и рекламы) «посмертной» отладки предоставляю анализ рассматриваемого дефекта в первоначальном варианте моего сообщения об ошибке¹ и предупреждаю о том, что он может быть не совсем понятен (если в нем вообще удастся разобраться):

[втс. 12/13/99]

Следующая последовательность событий может пояснить состояние дампа (стрелка указывает на их порядок):



¹ «Красивые сообщения об ошибках», я думаю, никто не против?

```

lwp_upimutex_lock() захватывает
upibr->upib_lock

lwp_upimutex_lock() видя, что
блокировка удерживается,
вызывает turnstile_block()

turnstile_block():
- Захватывает блокировку потока А
- Переводит А в TS_SLEEP
- Сбрасывает блокировку потока А
- Сбрасывает upibr->upib_lock
- Вызывает switch()

+-----+
| Держатель 0xff350000 отпускает блокировку, передавая ее явным
| образом потоку А (и устанавливая тем самым upi_owner на 300039c8580)
+-----+

Возвращается из turnstile_block()
    | Вызывает lwp_upimutex_lock()
    | во время блокировки 0xff350000
    |
    | lwp_upimutex_lock() захватывает
    | upibr->upib_lock
    |
    | Видя, что блокировка удерживается
    | (потоком А), вызывает
    | turnstile_block()

Вызывает lwp_upimutex_owned().
чтобы проверить передачу блокировки | | turnstile_block():
    | | - Захватывает блокировку потока Б
lwp_upimutex_owned() пытается | | - Переводит Б в TS_SLEEP,
захватить upibr->upib_lock | | устанавливая принадлежащий Б
                                | | wchan на upimutex,
                                | | соответствующий 0xff350000
upibr->upib_lock удерживается Б: | | - Пытается повысить уровень
вызывает в turnstile_block() | | держателя 0xff350000 (Поток А)
через mutex_vector_enter() | | - Захватывает блокировку потока А
turnstile_block(): | | - Корректирует приоритет А
                                | | - Сбрасывает блокировку потока А
+-----+
- Захватывает блокировку потока А | | - Сбрасывает блокировку потока Б
- Пытается повысить уровень
  держателя upibr->upib_lock
  (Поток Б)
- Захватывает блокировку потока Б | | - Сбрасывает upibr->upib_lock
- Корректирует приоритет Б
- Сбрасывает блокировку потока Б
- Видя, что принадлежащий Б wchan
  не равен NULL, пытается
  продолжить наследование

```

```

| приоритета          || |
| - Вызывает SOBJ_OWNER() в отношении | |
| wchan потока Б           ||           |
| - Видя, что владелецем      ||           |
| принадлежащего потоку Б wchan ||           |
| является поток А, выдает    ||           |
| тревожное сообщение:      ||           |
| "Взаимная блокировка:     ||           |
| цикл в блокирующей цепочке"||           |
|                                ||           |
+-----+ +-----+

```

Представленная последовательность наводит на размышления о том, что проблема кроется в `turnstile_block()`:

```

THREAD_SLEEP(t, &tc->tc_lock);
t->t_wchan = sobj;
t->t_sobj_ops = sobj_ops;
...
/*
 * Следование по блокирующей цепочке до ее окончания или до тех пор,
 * пока мы не выйдем из инверсии, завещая свой приоритет всем, кто
 * встречается на пути.
 */
while (inverted && t->t_sobj_ops != NULL &&
       (owner = SOBJ_OWNER(t->t_sobj_ops, t->t_wchan)) != NULL) {
    ...
(1) --> thread_unlock_nopreempt(t);
/*
 * На данный момент "t" может и не быть текущим потоком (curthread).
 * Поэтому с данного момента вместо "t" используется "curthread".
 */
if (SOBJ_TYPE(sobj_ops) == SOBJ_USER_PI) {
(2) -->     mutex_exit(mp);
    ...

```

Мы сбрасываем блокировку потока, относящуюся к заблокированному потоку (в точке (1)) перед тем, какбросить `upibr->upib_lock` в точке (2). От (1) до (2) мы нарушаём один из инвариантов блокировок `SOBJ_USER_PI`: в режиме ожидания на блокировке `SOBJ_USER_PI` не могут удерживаться никакие блокировки ядра: любые удерживаемые блокировки ядра могут привести к панике взаимной блокировки.

Чтобы разобраться в анализе, нужно кое-что знать о технических условиях реализации.

`upibr`

Указатель на внутреннее состояние ядра, связанный с удерживаемой блокировкой наследования приоритета пользовательского уровня; `upib_lock` — блокировка, защищающая это состояние.

`t_wchan`

Элемент структуры потока, содержащий указатель на примитив синхронизации, которым заблокирован поток (если таковой имеется)¹.

¹ `wchan` означает *wait channel* (ожидание канала), термин, относящийся к ранним временам Unix в Bell Labs, который сам по себе почти наверняка является незаконнорожденным производным от *event channels* (каналов событий) от Multics.

SOBJ_TYPE

Макрос, который берет направление операций для примитива синхронизации и возвращает константу, обозначающую тип; SOBJ_USER_PI – это константа, обозначающая блокировку, унаследовавшую приоритет на пользовательском уровне.

Суть проблемы состоит в следующем: для блокировок пользовательского уровня мы обычно отслеживаем состояние, связанное с блокировкой (например имеется или нет ожидающий поток) на уровне пользователя – ядром эта информация рассматривается исключительно в справочных целях. (Существует ряд ситуаций, при которых на бит ожидания нельзя полагаться и ядро знает, что не должно ему доверять в подобных ситуациях.)

Но для осуществления наследования приоритетов блокировок пользовательского уровня нужно проявлять особую щепетильность в отношении владельца, он должен отслеживаться в том же порядке, в каком мы отслеживали владельца для примитивов синхронизации на уровне ядра. Значит, когда в `turnstile_interlock()` мы затеваем сложный танец с блокировкой потока, то для определения владельца мы не должны осуществлять загрузку из памяти пользовательского уровня. Вся беда в том, что структура отслеживания владельца на пользовательском уровне, выстроенная на уровне ядра, сама должна быть защищена блокировкой, и эта блокировка (в ядре) должна сама осуществлять наследование приоритетов, чтобы избежать потенциальной инверсии.

Это ведет нас к непрогнозируемой взаимной блокировке: блокировка внутри ядра должна захватываться и сбрасываться, в том числе и для того чтобы захватить блокировку на пользовательском уровне и сбросить ее. Значит, возникают условия, при которых поток, владеющий блокировкой внутри ядра, хочет захватить блокировку на уровне пользователя, и возникают условия, при которых поток владеет блокировкой на пользовательском уровне и хочет захватить блокировку внутри ядра. В результате могут возникать зацикленные блокирующие цепочки, заставляющие ядро впадать в явную панику. И действительно, так именно и произошло в приведенном выше анализе отказа: поток А владел блокировкой на уровне пользователя и хотел захватить блокировку внутри ядра (`upib_lock`), а поток Б владел блокировкой внутри ядра и хотел завладеть блокировкой на уровне пользователя, а в результате мы получили взаимную блокировку!

Как только я понял суть проблемы, то смог без особого труда в точности ее воспроизвести: через пару минут я уже смог набросать контрольный пример, который вызвал панику системы тем же образом, что и в дампе. (В качестве отступления хочу заметить, что это одно из самых радостных чувств, возникающих при разработке программного обеспечения, когда в результате «посмертного» анализа отказа обнаруживаешь, что ошибка может быть легко воспроизведена, пишешь контрольный пример для проверки гипотезы, а затем наблюдаешь, что система рушится в точности, как тобою предсказано. Ничто не может всецело сравниться с этим чувством, для программиста это равноценно падению камня с души.)

Несмотря на то что у меня были некоторые идеи насчет устранения этой проблемы, отсрочка выпуска и серьезность сложившейся ситуации подтолкнули меня позвонить Джефу домой, чтобы обсудить проблему. Казалось, что при обсуждении проблемы мы с Джефом так и не могли придумать никакого решения, которое не вызывало бы новых проблем. И чем больше мы о ней говорили, тем сложнее она нам представлялась, и мы поняли, что допустили ошибку в самом начале, недооценив ее сложность и поручив решение задачи другому человеку.

Хуже того, мы с Джефом начали понимать, что у возникшей проблемы могут быть и другие скрытные проявления. Мы понимали, что если что-нибудь будет заблокировано на уровне ядра, то когда будет обнаружена мнимая взаимная блокировка, ядро явно запаникует. Но что получится, если что-нибудь будет заблокировано на пользовательском уровне, когда будет обнаружена мнимая взаимная блокировка? Мы довольно быстро определили (и подтвердили это контрольными примерами), что в таком случае попытка захватить блокировку на пользовательском уровне будет (ошибочно) возвращать EDEADLK. То есть ядро будет полагать, что «взаимная блокировка» вызвана примитивом синхронизации пользовательского уровня, и поэтому допускать, что взаимная блокировка была вызвана приложением, а именно закравшейся в него ошибкой.

Тем самым при данном виде отказа не имеющая ошибок программа получит при одном из своих вызовов `pthread_mutex_lock` ошибочный отказ – этот вид отказа имеет куда более серьезные последствия, чем паника, поскольку любое приложение, не проверяющее возвращаемое значение `pthread_mutex_lock` (а такое всегда найдется), может легко исказить свои собственные данные, полагая, что оно является владельцем блокировки, захват которой на самом деле не состоялся.

Эта проблема, если столкнуться с ней, не разбираясь в сути происходящего, практически не поддается отладке, и она, безусловно, подлежит устраниению.

Но как же решить все эти проблемы? Мы поняли, что легкого решения не предвидится, поскольку продолжали искать способы избежать подобной блокировки внутри ядра. Я представил себе блокировку внутри ядра как естественное ограничение в решении проблемы, но к этому заключению мы пришли крайне неохотно. Как только кто-нибудь из нас придумывал некую схему обхода этой блокировки, другой тут же находил какой-нибудь недостаток, сводящий ее на нет.

Измучив друг друга альтернативными вариантами, мы были вынуждены прийти к заключению, что блокировка внутри ядра ограничивала решение проблемы наследования приоритетов на пользовательском уровне, и наше внимание переключилось с обхода сложившейся ситуации на ее обнаружение.

Нужно было обнаружить два обстоятельства: возникновение паники и ложную взаимную блокировку. Обнаружить и обработать фальшивую блокировку не составляло особого труда, поскольку мы всегда оказывались в конце блокирующей цепочки и всегда обнаруживали, что блокировка, которой мы владеем, вызвавшая взаимную блокировку в блокировке внутри ядра, передана в виде параметра `turnstile_block`. Поскольку мы знаем, что завещали свой приоритет всей блокирующей цепочке, мы можем лишь определить это и выйти из цикла –

именно это и было описано в том путаном комментарии, который я добавил к `turnstile_block`, и именно этим занимались те две строки (блокировка внутри ядра, переданная в `turnstile_block`, хранится в локальной переменной `mp`).

С возникавшей паникой дело обстояло намного хуже. Хочу напомнить, что в этом случае поток владеет объектом синхронизации на пользовательском уровне и блокируется при попытке захвата блокировки внутри ядра. И в этом случае нам хотелось обойтись аналогичным способом: если взаимная блокировка заканчивается на текущем потоке, а последний поток в блокирующей цепочке заблокирован на объекте синхронизации пользовательского уровня, значит, взаимная блокировка является фальшивой. (То есть мы хотели бы справиться с этим обстоятельством за счет более общей обработки рассмотренного выше случая.) Все вроде бы просто, но опять-таки неприемлемо: при этом игнорируется возможность *реального возникновения* взаимной блокировки на уровне приложения (то есть ошибки приложения), при которой `EDEADLK` должен быть возвращен; нам требовался более выверенный подход.

Чтобы справиться с этим обстоятельством, мы обратили внимание на то, что если блокирующая цепочка шла от потоков, заблокированных на объекте синхронизации внутри ядра, до потоков, заблокированных на объектах синхронизации пользовательского уровня, мы знали, что это *не что иное*, как именно такой случай¹. Поскольку мы знали, что захватили другой поток в коде, в котором он не может получить приоритет (потому что знали, что в середине `turnstile_block` должен быть другой поток, который явным образом запрещает передачу приоритета), мы могли это исправить за счет ожидания снятия занятости, пока не произойдут изменения в блокировке, а затем перезапустить танец наследования приоритетов.

Вот как выглядит код для обработки подобных случаев²:

```
/*
 * Теперь у нас есть блокировка потока-владельца. Если мы перемещаемся от
 * не-SOBJ_USER_PI операций к SOBJ_USER_PI операциям, то в таком случае
 * мы знаем, что захватили поток, находящийся в состоянии TS_SLEEP.
 * но удерживающий mp. Мы знаем, что такая ситуация недолговечна
 * (mp будет сброшена перед тем, как держатель дезактивируется на объекте
 * синхронизации SOBJ_USER_PI), поэтому мы будем в цикле ожидать, пока mp
 * не будет сброшена. Затем, в случае отказа со стороны turnstile_interlock().
 * мы перезапустим танец наследования приоритетов.
 */
```

¹ Возможно, как и многие другие операционные системы, Solaris никогда не выполняет код на уровне пользователя с удержанием блокировок на уровне ядра и никогда не захватывает блокировки на пользовательском уровне из внутриддерных подсистем. Таким образом, этот случай единственный, при котором мы захватываем блокировку на уровне пользователя, удерживая блокировку на уровне ядра.

² Код, работающий с `turnstile_loser_lock`, на момент описания этого случая еще не существовал; он был добавлен для того, что бы справиться с другой (еще одной) проблемой, которую мы обнаружили в результате нашего совместного четырехдневного мозгового штурма. Эта проблема достойна отдельной главы, хотя бы за то звучное имя, которое ей присвоил Джейф: «дуэльные пораженцы». Вскоре после того как Джейф догадался о ее существовании, я обнаружил в действии ее вариант, который назвал «каскадные пораженцы». Но и дуэльным, и каскадным пораженцам придется дождаться лучших времен.

```
*/  
if (SOBJ_TYPE(t->t_sobj_ops) != SOBJ_USER_PI &&  
    owner->t_sobj_ops != NULL &&  
    SOBJ_TYPE(owner->t_sobj_ops) == SOBJ_USER_PI) {  
    kmutex_t *upi_lock = (kmutex_t *)t->t_wchan;  
  
    ASSERT(IS_UP1(upi_lock));  
    ASSERT(SOBJ_TYPE(t->t_sobj_ops) == SOBJ_MUTEX);  
  
    if (t->t_lockp != owner->t_lockp)  
        thread_unlock_high(owner);  
    thread_unlock_high(t);  
    if (loser)  
        lock_clear(&turnstile_loser_lock);  
  
    while (mutex_owner(upi_lock) == owner) {  
        SMT_PAUSE();  
        continue;  
    }  
    if (loser)  
        lock_set(&turnstile_loser_lock);  
    t = curthread;  
    thread_lock_high(t);  
    continue;  
}
```

Когда проблема была устранена, мы думали, что все уже позади. Но дальнейшее испытание под нагрузкой показало, что за всем этим скрывалась еще более глубокая проблема — одна из тех, которые, честно признаюсь, наводят на мысль о невозможности решения.

На сей раз признаки были другими: вместо вызова явной паники или неверного сообщения об ошибке система просто намертво зависала. Снятие (и изучение) дампа системы показало, что поток взаимно блокировался при попытке захватить блокировку потока из функции `turnstile_block()`, которая вызывалась рекурсивно из `turnstile_block()` через `mutex_vector_exit()`, функцию, которая освобождала мьютекс, если у того обнаруживались ожидающие. На основе лишь этого состояния проблема была ясна, и она ощущалась неким ударом ниже пояса.

Напомню, что эта коварная (но, к сожалению, необходимая) блокировка внутри ядра нуждается в захвате и сбросе как для захвата, так и для сброса блокировки наследования приоритета на уровне пользователя. Будучи заблокированной на блокировке пользовательского уровня, блокировка на уровне ядра должна быть сброшена после того, как поток передал в наследство свой приоритет в качестве последнего своего действия, перед тем как фактически уступить CPU посредством использования `swtch()`. (Это был код, процитированный, в частности, в моем первичном анализе; код, помеченный в нем как (2), является сбросом блокировки на уровне ядра.)

Но если другой поток заблокирован на уровне ядра в то время, как мы работаем с механикой блокировки, выставленной на уровне пользователя, мы должны будем активизировать ожидающего, сделав это составной частью сброса блокировки на уровне ядра. Пробуждение ожидающего требует получения

блокировки потока в таблице турникета, связанной с примитивом синхронизации, а затем, для отказа от любого унаследованного приоритета — захвата блокировки потока от прежнего держателя блокировки (то есть текущего потока).

Вот в чем проблема: мы входим в функцию, которая отказывается от унаследованного приоритета (в функцию `turnstile_pi_waive()`) из `turnstile_block()`, после того как мы уже оказались заблокированными. В частности, блокировка потока, относящаяся к текущему потоку, уже была изменена, и указывает не на блокировку текущего CPU, а на блокировку по записи в *таблице турникета*, которая соответствует блокировке на уровне пользователя, на котором мы фактически проводим блокировку. Поэтому если случится, что блокировка на уровне ядра и блокировка на уровне пользователя хэшируются на одной и той же записи в таблице турникета (что и произошло при отказе, в котором мы впервые с этим столкнулись), блокировка турникета, захватываемая в `turnstile_lookup()`, и блокировка потока, захватываемая в `turnstile_pi_waive()`, будут *одной и той же блокировкой* — и мы получим взаимную блокировку единственного потока. Даже если случится, что эти блокировки не будут хэшированы на одну и ту же запись в таблице турникета, но случится так, что они не будут придерживаться порядка блокировки, предписываемого `turnstile_interlock()`, мы получаем потенциальную возможность для классической взаимной блокировки АБ/БА. В любом случае это грязь.

Когда мы осознали суть проблемы, то нам показалось, что она не поддается решению. Учитывая, что основная проблема состояла в том, что мы сбрасывали блокировку внутри ядра после того, как оказывались заблокированными, весьма соблазнительно было бы найти способ исключения блокировки на уровне ядра. Но мы знали по работе с предыдущими ошибками, что подобные размышления вели в тупик; мы поняли, что блокировка на уровне ядра востребована, и знали, что она не должна быть сброшена, пока приоритет не будет завещан всей блокирующей цепочке.

Нам оставалось только пересмотреть более фундаментальные предположения: нельзя ли каким-то образом перевернуть порядок в `turnstile_block()`, чтобы мы завещали приоритет *перед* изменением структуры данных текущего потока, чтобы указать, что он находится в ожидании? (Нет, не получится, это откроет лазейку для инверсии приоритетов.) Не могли бы мы как-нибудь указать, что мы в таком состоянии, в котором вызов `turnstile_pi_waive()` из `turnstile_block()` через `mutex_vector_enter()` не вызывает взаимную блокировку? (Не подойдет, поскольку тогда мы не справимся с многопоточным сценарием взаимной блокировки.)

Какую бы гипотезу решения мы ни придумывали, практически сразу же видели ее фатальные недостатки и чем больше мы думали об этой проблеме, тем больше видели пороков, чем решений.

Повеяло безнадежностью; очень расстраивало, что простое добавление нового аспекта наследования приоритета на пользовательском уровне может свести на нет все то, что казалось вполне совершенным механизмом. Ложка превращалась в бочку, и мы чувствовали себя дрейфующими в сточных водах.

Когда мы собирались найти утешение в соседнем кафе, нас осенило: если наследование приоритетов на пользовательском уровне вызывало проблему,

то, может быть, мы рассуждали о ее решении слишком широко. Вместо того чтобы искать решение в наиболее абстрактной форме, почему бы, скажем, не справиться с этой проблемой за счет разбиения таблицы турникета? Мы можем хэшировать блокировки внутри ядра, защищающие состояние наследования приоритетов на пользовательском уровне в одной половине таблицы, и хэшировать любую другую блокировку в другой ее половине.

Это даст нам гарантии, что блокировка, которую мы сбросили бы немедленно, прежде чем вызвать `switch()` в `turnstile_block()`, *непременно* будет хэширована на другую запись в таблице турникета, а не на ту, к которой относится блокировка, которой мы были заблокированы. Кроме того, *гарантируя*, что любая блокировка на уровне ядра защищает состояние блокировки наследования приоритетов на пользовательском уровне, хешированной на запись таблицы турникета с более низким виртуальным адресом, чем любая запись таблицы турникета для любой другой разновидности блокировки, мы также даем гарантию, что последовательность блокировки, предписанная `turnstile_interlock()`, будет всегда соблюдаться; такое решение подошло бы как для однопоточных, так и для многопоточных случаев.

С одной стороны, это решение походило на некоторый довольно грубый частный случай, означающий помещение сведений об одной из специфических разновидностей блокировки (блокировки, защищающей внутри ядра состояние наследования приоритетов на пользовательском уровне) в универсальную систему турникетов. С другой стороны, мы были уверены, что все заработает и что это будет вполне разумная и практически лишенная риска доработка — и это обстоятельство играло весьма важную роль, если учесть, что шли последние дни двухлетнего цикла выпуска новой версии. Также было понятно, что у нас нет никаких других идей; и пока мы не придумаем что-либо более изящное, нужно было довольствоваться тем, что есть.

Итак, мы с Джефом за чашкой кофе обсудили детали нашего решения, и он вернулся, чтобы написать блок комментариев, поясняющий наши обманчиво простые изменения программного кода. Откровенно говоря, учитывая весьма спорную элегантность нашего решения, я ожидал, что комментарий будет похож на исповедь, приукрашенную обычными для таких комментариев определениями типа «грубое», «непривлекательное» или «отвратительное»¹. Но Джейф удивил меня тем, что я считаю наилучшим комментарием во всей операционной системе Solaris, если не во всем программном обеспечении:

/*
 * Хэш-таблица турникета разбита на две половины: нижняя из которых
 * используется для блокировок `iprimitextab[]`, а верхняя — для всех остальных.
 * Причина этих различий в том, что блокировки `SOBJ_USER_PI` преподносят
 * необычную проблему: блокировка `iprimitextab[]`, переданная `turnstile_block()`
 * не может быть сброшена до тех пор, пока вызываемый поток заблокирован на его
 * блокировке `SOBJ_USER_PI`, и завещает свой приоритет по нисходящей блокирующей
 * цепочке.
 * На данном этапе принадлежащий вызову `t_lockp` будет одной из блокировок

¹ Что натолкнуло на хороший совет: инцитте эти слова наряду с классическими зарезервированными словами типа «XXX» и «FIXME» — в любом исходнике, если хотите узнать, где собака зарыта.

```

* турникета.
* Если mutex_exit() обнаружит, что блокировка iprimitextab[] имеет ожидающих, она
* должна их активизировать, что вызывает для нас упорядочение блокировки:
* блокировка потока для блокировки iprimitextab[] будет захвачена в функции
* mutex_vector_exit(), которая в конечном итоге приведет в действие функцию
* turnstile_pi_waive(), которая затем захватит блокировку вызывающего потока.
* которая в данном случае будет блокировкой турникета для блокировки
* SOBJ_USER_PI. В общем, когда в одно и то же время должны удерживаться две
* блокировки турникета, блокировка должна быть упорядочена по адресам.
* Следовательно, для предотвращения взаимной блокировки в turnstile_pi_waive().
* мы должны сделать так, чтобы блокировки iprimitextab[] *всегда* хэшировались по
* более низким адресам по сравнению с любыми другими блокировками.
* Если вы считаете это решение неприглядным, то мы готовы посмотреть на ваш, более
* удачный вариант.
*/
#define TURNSTILE_HASH_SIZE 128 /* должен быть степенью числа 2 */
#define TURNSTILE_HASH_MASK (TURNSTILE_HASH_SIZE - 1)
#define TURNSTILE_SOBJ_HASH(sobj) \
    (((ulong_t)sobj >> 2) + ((ulong_t)sobj >> 9)) & TURNSTILE_HASH_MASK
#define TURNSTILE_SOBJ_BUCKET(sobj) \
    ((IS_UP(sobj) ? 0 : TURNSTILE_HASH_SIZE) + TURNSTILE_SOBJ_HASH(sobj))
#define TURNSTILE_CHAIN(sobj) turnstile_table[TURNSTILE_SOBJ_BUCKET(sobj)]

typedef struct turnstile_chain {
    turnstile_t      *tc_first:      /* первый турникет в хэш-цепочке */
    disp_lock_t      tc_lock:        /* блокировка для этой хэш-цепочки */
} turnstile_chain_t;

turnstile_chain_t      turnstile_table[2 * TURNSTILE_HASH_SIZE];

```

Тон, в котором Джейф выдержал свой комментарий, передал наши чувства намного точнее, чем та исповедь, которую я себе представлял: мы воплотили в жизнь это решение не потому, что потерпели поражение, а потому что это был единственный способ победить одну из наиболее сложных проблем, с которой каждый из нас когда-либо сталкивался. Кто-то может посчитать это решение неуклюжим, но за семь лет, прошедших с момента внедрения этого кода, никто не сделал ничего более подходящего — и поскольку я пишу об этом, то думаю, что не похоже, что кому-нибудь когда-нибудь удастся это сделать. По крайней мере я считаю, что код и не может быть красивее, при этом неважно, каким его считают, плохим или хорошим.

Итак, у нашей истории счастливый конец: мы внедрили доработки и выпустили продукт вовремя. Но полученный опыт служит нам напоминанием о некоторых принципах эффективной разработки программного обеспечения.

Раннее воплощение

Ни одну из проблем, с которыми нам пришлось столкнуться, мы с Джейфом не могли предвидеть, несмотря на то что мы оба потратили немало времени на обдумывание задачи в процессе ее проектирования и реализации. И даже когда мы обнаружили самые первые ошибки и подошли к проблеме вплотную, вся ее глубина для нас еще не раскрылась; чтобы понять, что происходит, нужно было непосредственно столкнуться со всеми этими неприятностями.

Усердие

Мы бы столкнулись с этими проблемами намного раньше, если бы специалист, которому было поручено решение задачи, провел нагрузочные испытания, вместо того чтобы полагаться исключительно на функциональные тесты. Разработчики программного обеспечения *несут ответственность за проведение своих собственных нагрузочных испытаний*. Тот, кто этого не придерживается, и в силу своих аристократических наклонностей считает, что написание подобных тестов – неподобающее занятие для его изнеженных рук истинного джентльмена от программирования, будет периодически поставлять никуда не годные программы. Это не означает, что не должно быть тестирующих или организаций, проводящих тестирование, просто тесты, созданные этими тестирующими и организациями, должны рассматриваться не в качестве замены, а в качестве дополнения к тестам, написанным первоначальными разработчиками.

Сосредоточение на экстремальных условиях

Стимул для начинающих разработчиков программного обеспечения вести отладку сложных систем стиснув зубы частично заключается в том, что тем самым прививаются навыки на всю оставшуюся жизнь: способность анализировать решение задачи в понятиях тех областей, в которых она не будет работать, вместо понятий о тех областях, где она может работать, дает возможность сосредоточиться на экстремальных условиях. Вынашивая планы создания новой программы, программисты не должны пытаться убеждать самих себя, по каким причинам эта конструкция будет работать, они должны устранить причины, по которым она работать не будет. Это не пропаганда суперанализа в противовес созданию кода, а скорее подсказка, что первый набросок программного кода любого проекта может содержать ошибки, которые способны свести на нет крупномасштабный конструкторский замысел.

Если эти принципы будут соблюдены, то появится естественное стремление разрешить все·наиболее серьезные проблемы на самой ранней стадии любого проекта и правильно разместить соответствующую инфраструктуру, чтобы проверить ее работоспособность и надежность. Без грязи, наверное, не обойдется, но вы получите гарантию, что самые противные ложки этой грязи будут перехвачены как можно раньше, когда еще можно вносить конструкторскую правку и когда еще можно будет сохранить в неприкосновенности бочку вина.

23 Распределенное программирование с MapReduce

Джеффри Дин (Jeffrey Dean) и Санджаи Гхемават (Sanjay Ghemawat)

В этой главе описывается конструкция и реализация MapReduce, системы программирования, решающей проблемы крупномасштабной обработки данных. MapReduce была разработана как способ упрощения разработки крупномасштабных вычислений в Google. Программы MapReduce автоматически распараллеливаются и выполняются на больших кластерах машин, имеющихся в продаже. Система поддержки выполнения отвечает за детали разбиения входящих данных, планирует выполнение программы на ряде машин, проводит обработку машинных ошибок и управление необходимым межмашинным общением. Это позволяет программистам, не имея никакого опыта работы с параллельными и распределенными системами, с легкостью использовать ресурсы большой распределенной системы.

Пример мотивации

Представьте, что в вашем распоряжении находится 20 миллиардов документов и вам нужно генерировать подсчет частоты появления в документах каждого уникального слова. При среднем размере документа 20 Кб только вычитка 400 терабайт данных на одной машине займет примерно четыре месяца. Если предположить, что мы согласны ждать столько времени и что у нас есть машина с достаточным объемом памяти, код будет относительно прост. В примере 23.1 (все примеры в этой главе относятся к псевдокоду) показан возможный алгоритм.

Пример 23.1. Простейшая программа подсчета слов непараллельным методом

```
map<string, int> word_count;
for each document d {
    for each word w in d {
        word_count[w]++;
    }
}
```

```

}
... сохранение word_count в постоянном хранилище ...

```

Один из способов ускорения этого вычисления — производство такого же вычисления параллельно для каждого отдельного документа, как показано в примере 23.2.

Пример 23.2. Распараллеленная программа подсчета слов

```

Mutex lock; // Защита word_count
map<string, int> word_count;
for each document d in parallel {
    for each word w in d {
        lock.Lock();
        word_count[w]++;
        lock.Unlock();
    }
}
... сохранение word_count в постоянном хранилище ...

```

Предыдущий код неплохо справляется с распараллеливанием программы со стороны ввода. В реальности код, зарождающий потоки, был бы немного более сложным, поскольку за счет использования псевдокода мы скрыли целый ряд деталей. Одной из проблем, связанных с примером 23.2, является использование единой глобальной структуры данных для отслеживания генерированных подсчетов. В результате узким местом, скорее всего, окажется существенная конкуренция по блокировке структуры данных word_count. Эта проблема может быть решена за счет разбиения структуры данных word_count на ряд сегментов с обособленной блокировкой на каждый сегмент, как показано в примере 23.3.

Пример 23.3. Распараллеленная программа подсчета слов с разделенным хранилищем

```

struct CountTable {
    Mutex lock;
    map<string, int> word_count;
};

const int kNumBuckets = 256;
CountTable tables[kNumBuckets];
for each document d in parallel {
    for each word w in d {
        int bucket = hash(w) % kNumBuckets;
        tables[bucket].lock.Lock();
        tables[bucket].word_count[w]++;
        tables[bucket].lock.Unlock();
    }
}
for (int b = 0; b < kNumBuckets; b++) {
    ... сохранение tables[b].word_count в постоянном хранилище ...
}

```

Программа все-таки еще имеет слишком упрощенный вид. Она не может быть масштабирована за пределы того количества процессоров, которое находится на отдельной машине. Наиболее доступные машины имеют не более восьми процессоров, так что даже при совершенном масштабировании этот подход потребует для завершения процесса нескольких недель.

Кроме этого, мы умолчали о проблеме, связанной с местом хранения входных данных и со скоростью их считывания отдельной машиной. Дальнейшее масштабирование требует, чтобы мы распределили данные и вычисления на несколько машин. Пока давайте предположим, что эти машины работают безотказно. Один из способов прироста масштабируемости заключается в запуске множества процессов на кластере связанных сетью машин. У нас будет множество процессов ввода, каждый из которых будет отвечать за чтение и обработку подмножества документов. У нас также будет и множество процессов вывода, каждый из которых будет отвечать за управление одним сегментом `word_count`. Этот алгоритм показан в примере 23.4.

Пример 23.4. Распараллеленная программа подсчета слов с разделенными процессорами

```

const int M = 1000; // Количество процессов ввода
const int R = 256; // Количество процессов вывода
main() {
    // Посчет количества документов, назначаемых каждому процессу
    const int D = number of documents / M;
    for (int i = 0; i < M; i++) {
        fork InputProcess(i * D, (i + 1) * D);
    }
    for (int i = 0; i < R; i++) {
        fork OutputProcess(i);
    }
    ... ожидание окончания всех процессов ...
}

void InputProcess(int start_doc, int end_doc) {
    map<string, int> word_count[R]; // Отдельная таблица для каждого
                                    // процесса вывода
    for each doc d in range [start_doc .. end_doc-1] do {
        for each word w in d {
            int b = hash(w) % R;
            word_count[b][w]++;
        }
    }
    for (int b = 0; b < R; b++) {
        string s = EncodeTable(word_count[b]);
        ... отправка s процессу вывода b ...
    }
}

void OutputProcess(int bucket) {
    map<string, int> word_count;
    for each input process p {
        string s = ... чтение сообщения от p ...
        map<string, int> partial = DecodeTable(s);
        for each <word, count> in partial do {
            word_count[word] += count;
        }
    }
    ... сохранение word_count в постоянном хранилище ...
}

```

Этот подход хорош для масштабирования на связанных сетью рабочих станций, но он намного сложнее и труднее в понимании (даже при том, что мы скрыли подробности маршализации и демаршализации, а также запуска и синхронизации различных процессов). Он также не в состоянии с честью преодолеть аппаратные сбои. Чтобы справиться с отказами, мы расширим пример 23.4, чтобы перед завершением заново выполнить процессы, потерпевшие крах. Чтобы избежать двойного подсчета данных, при повторном выполнении процесса ввода мы пометим каждую часть промежуточных данных сгенерированным номером процесса ввода, и изменим обработку вывода, чтобы в ней во избежание дубликатов использовались эти сгенерированные номера. Нетрудно представить, что с добавлением этой поддержки управления сбоями все становится еще сложнее.

Модель программирования MapReduce

Если сравнить пример 23.1 с примером 23.4, станет очевидным, что простая задача подсчета слов скрылась под массой деталей управления ее распараллеливанием. Если каким-то образом удалось бы отделить детали начальной задачи от деталей распараллеливания, то, может быть, удалось бы создать общую библиотеку распараллеливания или систему, применимую не только к задаче подсчета слов, но и к другим задачам, связанным с крупномасштабной обработкой данных. Используемый нами шаблон распараллеливания представляет собой следующее:

- для каждой входной записи извлекается набор пар ключ–значение, который мы обязательно получаем из каждой записи;
- для каждой извлеченной пары ключ–значение создается ее сочетание с другими значениями, использующими общий ключ (возможно, значения фильтрации, группировки или преобразования, используемые в процессе).

Давайте перепишем нашу программу, чтобы реализовать логику, отражающую специфику приложения подсчета частоты появления слов для каждого документа, и подведем итоги этого подсчета по документам в двух функциях, которые назовем *Map* и *Reduce*. Результат показан в примере 23.5.

Пример 23.5. Распределение задачи подсчета слов на функций Map и Reduce

```
void Map(string document) {  
    for each word w in document {  
        EmitIntermediate(w, "1");  
    }  
}  
  
void Reduce(string word, list<string> values) {  
    int count = 0;  
    for each v in values {  
        count += StringToInt(v);  
    }  
}
```

```

    Emit(word, IntToString(count));
}
}

```

Простая управляющая программа (драйвер), использующая эти функции для выполнения поставленной задачи на одной машине, будет выглядеть примерно так, как показано в примере 23.6.

Пример 23.6. Управляющая программа (драйвер) для Map и Reduce

```

map<string, list<string>> intermediate_data;

void EmitIntermediate(string key, string value) {
    intermediate_data[key].append(value);
}

void Emit(string key, string value) {
    ... запись пары ключ-значение в конечный массив данных ...
}

void Driver(MapFunction mapper, ReduceFunction reducer) {
    for each input item do {
        mapper(item)
    }
    for each key k in intermediate_data {
        reducer(k, intermediate_data[k]);
    }
}

main() {
    Driver(Map, Reduce);
}
}

```

Функция Map для каждой входной записи вызывается однократно. Любая промежуточная пара ключ–значение, выдаваемая функцией Map, непрерывно собирается кодом драйвера. Затем для каждого уникального промежуточного ключа вместе со списком связанных с этим ключом промежуточных значений вызывается функция Reduce.

Сейчас мы вернулись к той реализации, которая выполняется на одной машине. Но с подобным разделением мы теперь можем изменить реализацию программы драйвера, заставив ее работать с распределением, автоматическим распараллеливанием и отказоустойчивостью, не затрагивая логики, отражающей специфику приложения, которая заложена в функциях Map и Reduce. Кроме того, драйвер независим от специфической логики приложения, реализованной функциями Map и Reduce, и поэтому та же самая программа драйвера может быть повторно использована с другими функциями Map и Reduce для решения других задач. В заключение обратите внимание на то, что функции Map и Reduce, в которых реализована логика, отражающая специфику приложения, практически также понятны, как и простой последовательный код, показанный в примере 23.1.

Другие примеры MapReduce

Чуть позже мы исследуем реализацию намного более сложной программы драйвера, автоматически запускающей программы MapReduce на крупномасштаб-

ных кластерах машин, но сначала рассмотрим несколько других задач и путей их решения с использованием MapReduce.

Распределенное выполнение команды grep

Функция Map выдает строку, если она соответствует предоставленному шаблону регулярного выражения. Функция Reduce идентична функции, которая просто копирует предоставленные промежуточные данные на выход.

Обратный граф веб-ссылки

Препровождающий граф веб-ссылки – это граф, имеющий ребро от узла URL1 к узлу URL2, если веб-страница, найденная по URL1, имеет гиперссылку к URL2. Обратный граф веб-ссылки – это тот же граф, только с ребром, направленным в обратном направлении. MapReduce легко может быть использована для построения обратного графа веб-ссылки. Функция Map выдает пары *<цель, источник>* для каждой ссылки на целевой URL, найденной в документе названным источником. Функция Reduce связывает список всех URL источника с заданным целевым URL и выдает пары *<цель, список URL источника>*.

Вектор терминов в хосте

Вектор термина суммирует самые важные слова, встречающиеся в документе или подборке документов в виде списка пар *<слово, частота_появления>*. Функция Map выдает пары *<имя_хоста, вектор термина>* для каждого входящего документа (где имя_хоста извлекается из URL документа). Функция Reduce передает все относящиеся к документу векторы терминов заданному хосту. Он добавляет эти векторы терминов, отбрасывая редкие термины, а затем выдает конечную пару *<имя_хоста, вектор термина>*.

Инвертированный индекс

Инвертированный индекс представляет собой структуру данных, которая устанавливает соответствие каждого уникального слова списку документов, в котором оно встречается (где документы обычно идентифицируются с помощью цифрового идентификатора, чтобы инвертированные индексные данные сохраняли относительно малый объем). Функция Map анализирует каждый документ и выдает последовательность пар *<слово, id_документа>*. Функция Reduce принимает все id_документов для заданного слова, сортирует соответствующие документам ID и выдает пары *<слово, список_id_документов>*. Набор из всех выходных пар формирует простой инвертированный индекс. Это вычисление нетрудно пополнить, чтобы отслеживать позицию слова внутри каждого документа.

Распределенная сортировка

MapReduce также может быть использована для сортировки данных по определенному ключу. Функция Map извлекает ключ из каждой записи и выдает пару *<ключ, значение>*. Функция Reduce выдает все пары без изменений (например идентификационная функция Reduce). Это вычисление зависит от средств разделения и упорядочения свойств, описанных далее в этой главе.

Существует множество примеров вычислений, которые легко могут быть выражены через MapReduce. Что касается более сложных вычислений, то их зачастую нетрудно выразить как последовательность шагов MapReduce или как итерационное приложение MapReduce-вычисления, в котором выход одного шага MapReduce является входом для следующего шага MapReduce.

Как только вы начинаете думать о проблемах обработки данных в понятиях MapReduce, то выразить их чаще всего становится несколько легче. В качестве своеобразного доказательства можно отметить, что за последние четыре года количество MapReduce-программ в Google прошло путь от небольшой горстки претендентов на решение задачи в марте 2003 года (когда мы приступили к конструированию MapReduce) до более чем 6000 отдельных программ MapReduce в декабре 2006 года. Эти программы были написаны более чем тысячью разных разработчиков программного обеспечения, многие из которых до использования MapReduce никогда раньше не создавали параллельных или распределенных программ.

Распределенная реализация MapReduce

Наибольшие преимущества от программирования с использованием модели MapReduce состоят в том, что она превосходно отделяет выражение требуемого вычисления от лежащих в основе подробностей распараллеливания, обработки отказов и т. п. Разумеется, для разных видов компьютерных платформ существуют различные реализации моделей программирования MapReduce. Правильный выбор зависит от окружающей обстановки. К примеру, одна реализация может подходить для небольших машин с общим пространством памяти, другая — для больших NUMA-мультипроцессоров, а еще одна для еще большей совокупности сетевых машин.

Самая простая реализация, рассчитанная на работу с единственной машиной и поддерживающая модель программирования, была показана в кодовом фрагменте, приведенном в примере 23.6. В этом разделе описывается более сложная реализация, нацеленная на крупномасштабную работу MapReduce в вычислительной среде, которая широко используется в компании Google: больших кластерах имеющихся в широкой продаже персональных компьютеров, объединенных в сеть коммутируемым Ethernet (см. раздел «Дополнительная информация», который находится в конце этой главы). В этой среде:

- машины представляют собой обычные двухпроцессорные вычислительные блоки x86, запущенные под управлением Linux с 2–4 Гб памяти на каждую машину;
- машины связаны между собой с использованием имеющегося в широкой продаже сетевого оборудования (обычно это коммутаторы Ethernet с пропускной способностью 1 Гбит в секунду). Машины смонтированы в стойках по 40 или 80 единиц. Эти стойки присоединены к центральному коммутатору, работающему на весь кластер. Полоса пропускания, доступная при обмене данными с машинами того же кластера, составляет 1 Гб/с на машину, в то время как полоса пропускания, приходящаяся на одну машину с использованием центрального коммутатора, намного скромнее (обычно от 50 до 100 Мбит/с на машину).

- о хранение данных реализовано на недорогих IDE-дисках, подключенных непосредственно к отдельным машинам. Распределенная файловая система, названная GFS (см. ссылку на «The Google File System» в разделе «Дополнительная информация» в конце этой главы), используется для управления данными, хранящимися на этих дисках. GFS использует дублирование для того, чтобы обеспечить доступность и надежность, компенсируя отказы оборудования за счет разбиения файлов на участки по 64 Мб и хранении (как правило) трех копий каждого участка на разных машинах;
- о пользователи выдают задание системе планирования. Каждое задание состоит из набора задач и распределяется планировщиком по набору доступных в кластере машин.

Общее представление о выполнении

Вызовы функции Map распределяются по нескольким машинам за счет автоматического разделения входных данных на ряд M-частей. Входные части могут быть обработаны параллельно на разных машинах. Вызовы функции Reduce распределены за счет деления промежуточного ключевого пространства на R-части с использованием функций разбиения (например $\text{хэш}(ключ) \% R$).

На рис. 23.1 показаны действия, выполняемые, когда пользовательская программа вызывает функцию MapReduce (числовые метки на рис. 23.1 соответствуют нумерации в следующем списке).

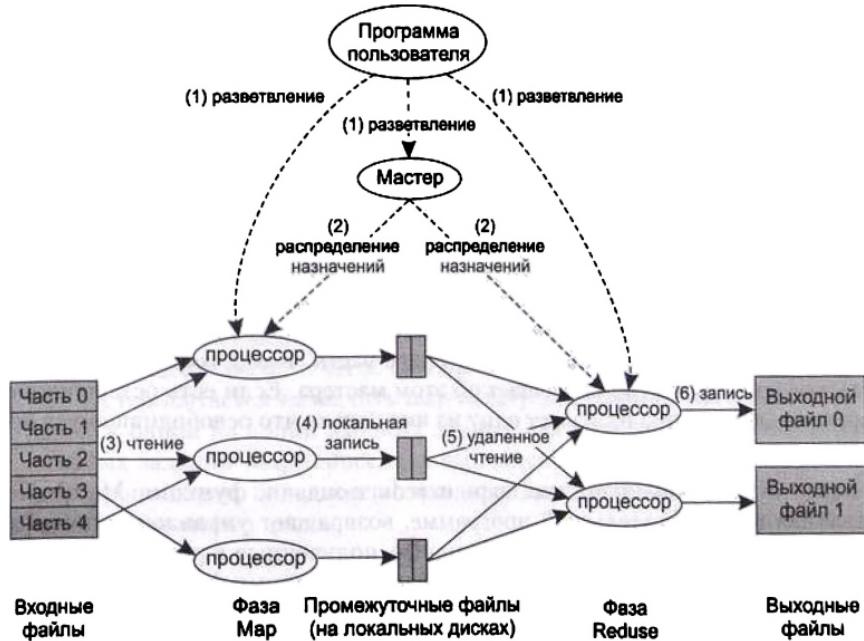


Рис. 23.1. Взаимоотношения между процессами в MapReduce

- Сначала библиотека MapReduce разбивает входные файлы на M-частей (обычно от 16 до 64 Мб на каждую часть). Затем за счет создания запроса к кластерной системе планирования на кластере машин запускается множество копий программы.
- Одна из копий имеет специальное назначение и называется мастером MapReduce. Остальные задачи представляют собой назначенные мастером части Map- и Reduce-работы. Есть M map-задач и R reduce-задач. Мастер задействует свободные рабочие процессоры и назначает каждому из них map- и (или) reduce-задачу.
- Процессор, которому назначена map-задача, считывает содержание соответствующей входной части. Он передает каждую входящую запись определенной пользователем Map-функции. Промежуточные пары ключ–значение, выработанные Map-функцией, буферизируются в памяти.
- Периодически буферизированные пары записываются на локальный диск, из которых функция разбиения составляет R отдельных частей. Когда map-задача будет завершена, процессор уведомляет об этом мастера. Мастер передает информацию о размещении промежуточных данных, сгенерированную map-задачей другим процессорам, которые были назначены для выполнения reduce-задач. Если есть оставшиеся map-задачи, мастер назначает одну из оставшихся задач только что освободившемуся процессору.
- Когда reduce-процессору сообщается местоположение промежуточных данных для выполнения назначенному ему reduce-задачи, он выполняет удаленные вызовы процедуры для чтения буферизированных промежуточных данных с локальных дисков тех процессоров, которые выполняли map-задачи. Когда reduce-процессор завершает чтение всех промежуточных данных для выполнения своей reduce-задачи, он сортирует их по промежуточным ключам, чтобы они были сгруппированы вместе по совпадению этих ключей. Если промежуточные данные слишком велики, чтобы поместиться в памяти reduce-процессора, используется внешняя сортировка.
- Reduce-процессор осуществляет перебор отсортированных промежуточных пар ключ–значение. Для каждого обнаруженного уникального промежуточного ключа он передает ключ и соответствующий ему список промежуточных значений пользовательской Reduce-функции. Любые пары ключ–значение, сгенерированные пользовательской Reduce-функцией, добавляются к конечному выходному файлу данной reduce-части. Когда reduce-задача будет выполнена, процессор оповещает об этом мастера. Если есть оставшиеся reduce-задачи, мастер назначает одну из них только что освободившемуся процессору.

Когда будут выполнены все map- и reduce-задачи, функция MapReduce, вызванная в пользовательской программе, возвращает управление пользовательскому коду. К этому времени выходные, полученные в результате работы MapReduce, доступны в R выходных файлов (по одному файлу на reduce-задачу).

Некоторые детали реализации позволяют ей успешно выполняться в созданной нами среде.

Выравнивание нагрузки

Обычно у задания MapReduce намного больше задач, чем машин, а это означает, что каждому процессору мастер назначает множество различных задач. Новая задача машине назначается мастером, когда будет выполнена предыдущая задача. Это означает, что более быстрая машина получит больше задач, чем та, которая работает медленнее. Следовательно, назначение машинам задач позволяетенным образом выровнять нагрузку даже в неоднородной среде, и в течение всего процесса вычислений поддерживается постоянная загруженность процессоров полезной работой.

Отказоустойчивость

Поскольку эта реализация MapReduce спроектирована для выполнения заданий, распределенных среди сотен или тысяч машин, библиотека должна вполне очевидно справляться с отказами машин. Мастер хранит информацию о состоянии той или иной задачи map и reduce, выполняемой процессором. Периодически мастер вызывает на каждом процессоре удаленную процедуру ping. Если процессор не отвечает на несколько последовательно вызванных процедур, мастер считает, что процессор неисправен, и назначает всю работу, которая выполнялась этим процессором, другим машинам для их повторного выполнения. Поскольку при обычном выполнении MapReduce map-задач может быть в 50 раз больше, чем процессоров, восстановление происходит очень быстро из-за того, что из 50 отдельных машин при отказе одной из них любая из оставшихся может взять на себя ее повторное выполнение. Мастер регистрирует все обновления в состоянии своего планирования в постоянном файле регистрационного журнала. Если мастер дает сбой (что случается крайне редко, поскольку мастер всего один), он перезапускается кластерной системой планирования. Новый экземпляр мастера считывает файл журнала, восстанавливая прежнее внутреннее состояние.

Размещение

Наша реализация MapReduce сохраняет полосу пропускания сети за счет преимуществ, обусловливаемых тем фактом, что входные данные (управляемые GFS) хранятся на тех же самых машинах или стойках, на которых выполняются map-вычисления. Для каждой имеющейся задачи Map мастер MapReduce определяет местонахождение входных данных (обычно из-за дублирования, осуществляемого GFS, они имеют множественное размещение). Затем мастер пытается назначить map-задачу той машине, которая находится ближе к одной из копий входных данных для этой задачи. Для крупномасштабных заданий MapReduce, для выполнения которых задействуются тысячи процессоров, большинство входных данных считаются непосредственно с локального диска.

Дублирование

Время работы MapReduce зачастую зависит от нескольких отстающих машин. (Отстающей может быть любая машина, которая довольно долго выполняет одну из последних map- или reduce-задач.) Задача может долго

выполняться, либо потому что она сама является трудоемкой, либо потому что ее выполнение запущено на медленной машине.

Машину может быть медленной по многим причинам. Например, она может быть занята другим посторонним процессом, сильно загружающим процессор, или у машины может быть дефектный жесткий диск, из-за которого часто происходит перечитывание информации, замедляющее его работу в десять или в сто раз.

Чтобы решить проблему с отстающими, мы используем дублирование. Когда остается всего несколько тар-задач, мастер назначает незадействованным процессорам по одному дублирующему выполнению каждой оставшейся из еще выполняемых тар-задач. Каждая оставшаяся тар-задача помечается как завершенная, как только одно из ее решений будет закончено (первоначальное или дублирующее). Подобная стратегия используется и для reduce-задач. Для выполнения дублирующих задач обычно задействуется 1–2 % дополнительных вычислительных ресурсов, но оказывается, что они значительно сокращают время завершения крупномасштабных MapReduce-операций.

Расширения модели

Хотя в большинстве своих применений MapReduce требует только написания функций Map и Reduce, мы расширили базовую модель за счет некоторых свойств, которые посчитали практически полезными.

Функция разбиения

Пользователи MapReduce определяют желаемое количество выходных файлов reduce-задач (R). Промежуточные данные разбиваются на эти задачи функцией разбиения по промежуточному ключу. Предоставляемая по умолчанию функция разбиения для равномерного распределения данных по R -частям использует хэширование ($\text{хэш}(ключ) \% R$).

Но в некоторых случаях куда полезнее разбивать данные с помощью какой-нибудь другой функции ключа. К примеру, иногда выходными ключами служат URL, и нам нужно, чтобы все вхождения на отдельный хост оказались в одном выходном файле. Чтобы отвечать ситуации подобной этой, пользователи библиотеки MapReduce могут предоставить свои собственную функцию разбиения. К примеру, использование в качестве такой функции $\text{хэш}(\text{имя_хоста}(\text{url_ключ})) \% R$ приведет к тому, что все URL одного хоста будут сведены в один выходной файл.

Гарантии упорядоченности

Наша реализация MapReduce сортирует промежуточные данные в группы, в которых собираются все промежуточные значения, использующие один и тот же промежуточный ключ. Поскольку многие пользователи считают удобным вызывать свою функцию Reduce по отсортированным ключам, и мы уже сделали все необходимое, мы раскрываем это перед пользователями, га-

рантируя это свойство упорядочения в интерфейсе к библиотеке MapReduce.

Пропуск плохих записей

Иногда в пользовательском коде встречаются ошибки, приводящие к неизбежным сбоям функций Map или Reduce на конкретных записях. Подобные ошибки могут приводить к провалу выполнения крупномасштабных заданий MapReduce, после того как уже будет проделана значительная часть вычислений. Наиболее предпочтительным направлением действий будет устранение ошибки, но иногда последовать ему не представляется возможным; к примеру, ошибка может быть в какой-нибудь библиотеке стороннего производителя, и ее исходный код может быть недоступен. К тому же иногда вполне приемлемо проигнорировать несколько записей, к примеру, при проведении статистического анализа большого набора данных. Поэтому мы предоставили дополнительный режим выполнения, при котором библиотека MapReduce определяет записи, вызывающие неизбежный отказ, и пропускает их обработку в последующих повторных выполнениях в целях продолжения работы.

Для каждой задачи процессора устанавливается обработчик сигнала, отлавливающий нарушения сегментации и ошибки шины. Перед вызовом пользовательских операций Map или Reduce библиотека MapReduce сохраняет порядковый номер записи в глобальной переменной. Если пользовательский код генерирует сигнал, обработчик сигнала отправит UDP-пакет «последнего вздоха», в котором будет содержаться порядковый номер, предназначенный для мастера MapReduce. Когда мастер отследит на конкретной записи более одной ошибки при решении вопроса о следующем повторном выполнении соответствующей задачи Map или Reduce, он отметит, что эта запись должна быть пропущена.

Ряд других расширений рассматривается в подробной статье о MapReduce (см. далее раздел «Дополнительная информация»).

Заключение

MapReduce доказала свою ценность в Google. На начало 2007 года у нас было более чем 6000 отдельных программ, использующих модель программирования MapReduce, и запускалось более 35 000 заданий MapReduce в день, в ходе которых за день обрабатывалось около 8 петабайт входных данных (поддерживающаяся скорость составляла около 100 гигабайт в секунду). Хотя изначально мы разрабатывали модель программирования MapReduce как часть нашей работы по переписыванию системы индексирования для нашего поискового веб-сайта, она показала себя полезной в очень широком спектре задач, включая машинное обучение, статистический машинный перевод, анализ регистрационных записей, эксперименты по информационному поиску, а также общую крупномасштабную обработку данных и проведение объемных вычислений.

Дополнительная информация

Более подробное описание MapReduce было представлено на конференции OSDI '04:

«MapReduce: Simplified Data Processing on Large Clusters». Джейфри Дин (Jeffrey Dean) и Санджай Гхемават (Sanjay Ghemawat). Представлено на OSDI '04: 6-й симпозиум по проектированию и реализации операционных систем, Сан-Франциско, Калифорния, декабрь 2004 года. Доступно по адресу: <http://labs.google.com/papers/mapreduce.html>.

Статья о проектировании и реализации файловой системы Google File System, представленная на конференции SOSP '03:

«The Google File System». Санджай Гхемават (Sanjay Ghemawat), Ховард Гобиоф (Howard Gobioff) и Шунь-Так Лэнг (Shun-Tak Leung). 19-й ACM симпозиум по основам операционных систем, Лейк Джордж, Нью-Йорк, октябрь 2003 года. Доступна по адресу: <http://labs.google.com/papers/gfs.html>.

Статья, описывающая основную аппаратную инфраструктуру Google, представленная на IEEE Micro:

«Web Search for a Planet: The Google Cluster Architecture». Луис Баррозо (Luiz Barroso), Джейфри Дин (Jeffrey Dean) и Урс Хёлзле (Urs Hölzle). IEEE Micro, том 23, номер 2 (март 2003 г.), с. 22–28. Доступна по адресу: <http://labs.google.com/papers/googlecluster.html>.

Язык, названный Sawzall, разработанный Google для анализа регистрационных записей, запускаемый на основе MapReduce:

«Interpreting the Data: Parallel Analysis with Sawzall». Роб Пайк (Rob Pike), Шон Дорвард (Sean Dorward), Роберт Гриземер (Robert Griesemer) Шон Куинлан (Sean Quinlan). Scientific Programming Journal Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure 13:4, pp. 227–298. Доступно по адресу: <http://labs.google.com/papers/sawzall.html>.

Благодарности

В дальнейшее развитие и усовершенствование MapReduce был внесен существенный вклад со стороны ряда специалистов, в числе которых Том Анно (Tom Apna), Мэтт Остерн (Matt Austern), Крис Колохан (Chris Colohan), Франк Дабек (Frank Dabek), Уолт Друммонд (Walt Drummond), Сяньпин Гэ (Xianping Ge), Виктория Гилберт (Victoria Gilbert), Шань Лэй (Shan Lei), Джош Левенберг (Josh Levenberg), Нахуш Махаджан (Nahush Mahajan), Грэг Малевич (Greg Malewicz), Рассел Пауэр (Russell Power), Уилл Робинсон (Will Robinson), Иоаннис Цукалидис (Ioannis Tsoukaliidis) и Джерри Чжао (Jerry Zhao). MapReduce строится на основе некоторых составляющих инфраструктуры, разработанной в Google, включая файловую систему Google File System и нашу систему кластерного распределения. Хочется выразить особую благодарность разработчикам этих систем. И в завершение мы благодарим всех пользователей Мар-

Reduce, относящихся к техническим работникам Google за предоставление полезных отзывов, предложений и сообщениях об ошибках.

Приложение: Решение задачи подсчета слов

В этом разделе содержится пример полного решения примера по подсчету частоты появления слов, реализованного на C++, о котором упоминалось в самом начале этой главы. Этот код также можно найти на веб-сайте издательства O'Reilly, созданном для этой книги (<http://www.oreilly.com/catalog/9780596510046>):

```
#include "mapreduce/mapreduce.h"

// пользовательская map-функция
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Пропуск лидирующих пробелов
            while ((i < n) && isspace(text[i]))
                i++;

            // Обнаружение окончания слова
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
            if (start < i)
                EmitIntermediate(text.substr(start,i-start), "1");
        }
    }
};

REGISTER_MAPPER(WordCounter);

// Пользовательская reduce-функция
class Adder : public Reducer {
public:
    virtual void Reduce(ReduceInput* input) {
        // Итерация по всем входам с тем же самым
        // ключом и добавление значений
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Выдача суммы для input->key()
        Emit(IntToString(value));
    }
};

REGISTER_REDUCER(Adder);
```

```
int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Сохранение списка входных файлов в "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Определение выходных файлов:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Дополнительно: частичное суммирование в рамках
    // map-задач с целью сохранения пропускной способности сети
    out->set_combiner_class("Adder");

    // Настроочные параметры: использование не более 2000 машин
    // и 100 Мб памяти на каждую задачу
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Теперь выполнение всего этого
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Готово: структура 'result' содержит информацию
    // о счетчиках, затраченном времени, количестве
    // задействованных машин и т. д.
    return 0;
}
```

24 Красота параллельной обработки

Саймон Пейтон Джонс (*Simon Peyton Jones*)

Бесплатные обеды закончились¹. Мы привыкли к мысли, что наши программы будут работать быстрее, если купим процессор следующего поколения, но это время прошло. Хотя на новом кристалле будет больше центральных процессоров, каждый отдельный CPU не будет быстрее прошлогодней модели. Если мы хотим, чтобы наши программы работали быстрее, нужно научиться писать параллельные программы².

Параллельные программы выполняются неопределенным образом, поэтому их трудно тестировать, а дефекты практически невозможно воспроизвести. Я считаю, что красивая программа отличается простотой и изяществом, не потому что в ней нет явных ошибок, а потому что в ней явно не может быть никаких ошибок³. Если мы хотим создавать надежно работающие параллельные программы, то особое внимание нужно уделять их красоте. К сожалению, параллельные программы зачастую менее красивы, чем их последовательные родственницы; в частности, как мы увидим, они менее *модульны*.

В этой главе я дам описание *программной транзакционной памяти* – *Software Transactional Memory* (STM), многообещающему новому подходу к программированию параллельных процессоров с общим пространством памяти, который, как представляется, поддерживает модульные программы, используя метод, недоступный текущим технологиям. К тому времени, когда мы завершим рассмотрение этого вопроса, я надеюсь, что вы станете таким же сторонником STM, как и я. Конечно, эта технология – не решение всех проблем, но все же она представляет собой красивую и воодушевляющую атаку на пугающие своей мощью крепостные валы параллельной обработки информации.

¹ Herb Sutter, «The free lunch is over: a fundamental turn toward concurrency in software», Dr. Dobb's Journal, March 2005.

² Herb Sutter, James Larus, «Software and the concurrency revolution», ACM Queue, vol. 3, № 7, Sep. 2005.

³ Это высказывание принадлежит Тони Хоаре (Tony Hoare).

Простой пример: банковские счета

Возьмем простую задачу по программированию.

Напишите процедуру для перевода денег с одного банковского счета на другой. Чтобы не усложнять задачу, оба счета хранятся в памяти: то есть не требуется никакого взаимодействия с базами данных. Процедура должна правильно работать в параллельной программе, в которой множество потоков могут вызывать перевод одновременно. Ни один из потоков не должен наблюдать картину, при которой деньги ушли с одного счета, но так и не пришли на другой (или наоборот).

Это несколько отвлеченный пример, но его простота позволяет нам сосредоточиться в этой главе на новизне решения: на языке Haskell и транзакционной памяти. Но сначала давайте взглянем на традиционный подход.

Банковские счета, использующие блокировки

Доминирующей технологией, используемой сегодня для согласования работы параллельных программ, является использование *блокировок* и *переменных условий*. В объектно-ориентированных языках неявная блокировка, осуществляющаяся за счет *синхронизированных методов*, есть у каждого объекта, при этом используется та же идея. Поэтому класс для банковских счетов можно было бы определить следующим образом:

```
class Account {
    Int balance;
    synchronized void withdraw( Int n ) {
        balance = balance - n;
    }
    void deposit( Int n ) {
        withdraw( -n );
    }
}
```

При использовании метода `synchronized` для снятия средств — `withdraw` следует соблюдать осторожность, чтобы не получилось незамеченных уменьшений счета, если два потока вызовут `withdraw` одновременно. Действие `synchronized` заключается в блокировке счета, запуске `withdraw` и последующем снятии блокировки.

Код для перевода средств можно было бы написать следующим образом:

```
void transfer( Account from, Account to, Int amount ) {
    from.withdraw( amount );
    to.deposit( amount );
}
```

Такой код вполне подошел бы для последовательной программы, но в параллельной другой поток может попасть в промежуточное состояние, в котором деньги уже ушли со счета `from` но не пришли на счет `to`. То, что оба метода синхронизированы — `synchronized`, нам не поможет. Счет `from` сначала блокируется, а потом разблокируется за счет вызова метода `withdraw`, а затем он блокируется и разблокируется методом `deposit`. Очевидно, что между двумя вызовами деньги отсутствуют на обоих счетах.

Для финансовых программ это обстоятельство может быть неприемлемым. Как исправить положение? Обычно применяется решение, при котором добавляется код явной блокировки:

```
void transfer( Account from, Account to, Int amount ) {  
    from.lock(); to.lock();  
    from.withdraw( amount );  
    to.deposit( amount );  
    from.unlock(); to.unlock(); }
```

Но эта программа обладает фатальной склонностью к взаимной блокировке. Рассмотрим, в частности, маловероятную ситуацию, при которой другой поток переводит деньги в обратном направлении между этими же двумя счетами. Тогда каждый поток может вызвать одну блокировку, а затем заблокироваться, бесконечно ожидая действий другого потока.

Когда эта не всегда очевидная проблема обнаружится, стандартным выходом из нее будет установка произвольного глобального порядка блокировок и применение их по возрастающей. При этом код блокировки приобретает следующий вид:

```
if from < to  
    then { from.lock(); to.lock(); }  
else { to.lock(); from.lock(); }
```

Если весь набор блокировок может быть заранее предсказан, то все работает просто превосходно, но такое удается не всегда. Представим, к примеру, что `from.withdraw` выполняется для перевода средств со счета `from2`, если их не хватает на счете `from`. Мы не знаем, нужно ли осуществлять блокировку `from2`, пока не считано состояние счета `from`, а потом ставить блокировку в «правильной» последовательности уже поздно. Кроме того, само существование `from2` может быть конфиденциальной информацией, известной счету `from`, но неизвестной системе перевода `transfer`. И даже если система перевода действительно знала о счете `from2`, то теперь блокирующий код должен выставить три блокировки, распределив их по возможности в правильном порядке.

Дело еще больше усложняется, когда мы хотим заблокировать действие. Предположим, к примеру, что перевод должен блокироваться, если на счете `from` недостаточно средств. Обычно это делается за счет обслуживания *переменной условий* при одновременном снятии блокировки со счета `from`. Все еще больше усложняется, если мы хотим удерживать блокировку, пока не будет достаточно средств на счетах `from` и `from2`, рассматриваемых вместе.

Порочность блокировок

Короче говоря, доминирующая на сегодняшний день технология параллельного программирования — блокировки и переменных условий — никуда не годится. Вот как выглядит ряд присущих ей типовых пороков, часть из которых мы уже рассмотрели.

Выставление недостаточного количества блокировок

Можно забыть выставить блокировку и получить в результате два потока, одновременно изменяющих одну и ту же переменную.

Использование излишних блокировок

Можно выставить слишком много блокировок и свести на нет весь эффект параллельной обработки (в лучшем случае) или вызвать взаимную блокировку (в худшем случае).

Использование неверных блокировок

При программировании на основе блокировок связь между блокировкой и защищаемыми ею данными зачастую существует лишь в воображении программиста и не прослеживается в явном виде в самой программе. Как следствие, довольно просто выставить или удерживать неверную блокировку.

Использование блокировок в неверной очередности

При программировании с использованием блокировок нужно позаботиться о выставлении блокировок в «правильной» последовательности. Устранение взаимной блокировки, которая может произойти в противном случае, — занятие довольно утомительное, не гарантированное от ошибок и порой невероятно сложное.

Восстановление работоспособности после возникновения ошибки

Восстановление работоспособности может быть затруднено, поскольку программист должен гарантировать, что в системе, состояние которой непостоянно или в которой блокировки удерживаются неопределенным образом, не должно оставаться ошибок.

Пропущенные активизации и ошибочные повторения

Можно забыть подать сигнал с помощью переменной условий, которого ожидает поток, или перепроверить условие после активизации.

Но главный недостаток программирования на основе блокировок заключается в том, что блокировки и переменные условий не поддерживают модульное программирование. Под «модульным программированием» я подразумеваю процесс построения больших программ путем объединения более мелких программ. Блокировки делают это невозможным. К примеру, мы не могли использовать наши (вполне корректные) реализации снятия — `withdraw` и зачисления — `deposit` средств в неизменном виде для того, чтобы реализовать перевод — `transfer`; вместо этого нам пришлось предоставить протокол блокировки. Блокировка и выбор обладают еще меньшей модульностью. Предположим, к примеру, что у нас была версия `withdraw`, которая блокировалась, если на счете-источнике было недостаточно средств. Тогда мы не сможем воспользоваться `withdraw` напрямую для снятия средств со счета А или Б (в зависимости от того, на котором из них достаточно средств) без выставления условий блокировки, но даже при этом задача будет не из легких. Эти критические замечания детально изложены в других источниках¹.

¹ Edward A. Lee, «The problem with threads», IEEE Computer, vol. 39, № 5, pp. 33–42, May 2006 г.; J. K. Ousterhout, «Why threads are a bad idea (for most purposes)», Invited Talk, USENIX Technical Conference, Jan. 1996 г.; Tim Harris, Simon Marlow, Simon Peyton Jones, «Composable memory transactions», ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005.

Программная транзакционная память

Программная транзакционная память (STM) – это многообещающий новый подход к решению проблем параллельной обработки, который будет рассмотрен в этом разделе. Для объяснения сути STM будет использован Haskell, наиболее красивый из всех известных мне языков программирования, поскольку STM вписывается в Haskell с особым изяществом. Если вы не знакомы с Haskell, ничего страшного; мы освоим его в процессе чтения раздела.

Побочные эффекты и ввод–вывод в языке Haskell

Вот как начинается код перевода средств на языке Haskell:

```
transfer :: Account -> Account -> Int -> IO ()
-- Перевод 'amount' со счета 'from' на счет 'to'
transfer from to amount = ...
```

Вторая строка этого определения, начинающаяся с группы символов --, является комментарием. В первой строке задается *сигнатурой типа* для transfer¹. Эта сигнатура объявляет, что transfer воспринимает в качестве своих аргументов два значения типа Account (счет-источник и счет-приемник) и одно значение типа Int (сумма перевода) и возвращает значение типа IO (). Этот тип результата сообщает, что «перевод (transfer) возвращает действие, которое при выполнении может иметь ряд побочных эффектов, а затем возвращает значение типа ()». Тип () произносится «юнит» (unit), имеет только одно значение, которое тоже записывается как (); он сродни типу void в языке Си. Итак, тип результата transfer – IO () объявляет, что его побочный эффект составляет единственную причину для его вызова. Перед тем как пойти дальше, нужно объяснить, как в Haskell обрабатываются побочные эффекты.

Побочным эффектом считается все, что считывает или записывает изменяющее состояние. Яркими примерами побочного эффекта являются операции ввода–вывода (Input/output). Вот, к примеру, как выглядят сигнатуры двух функций Haskell, обладающих эффектами ввода–вывода:

```
hPutStr :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
```

Любое значение типа IO t мы называем *действием*. Значит, (hPutStr h «hello») – это действие² которое при выполнении выведет hello на описатель

¹ Может показаться странным, что в этой сигнатуре типа присутствуют три функциональные стрелки, а не одна. Это связано с тем, что Haskell поддерживает карринг (currying), который описывается в любой книге по Haskell («Haskell: The Craft of Functional Programming», S. J. Thompson, Addison-Wesley) или в Википедии. Чтобы усвоить материал этой главы, просто рассматривайте все типы, кроме последнего, в качестве аргументов.

² В Haskell для записи прикладной функции используется простое непосредственное размещение. В большинстве языков вы напишете hPutStr(h, "hello"), но в Haskell запись выглядит проще: (hPutStr h "hello").

(handle¹) h и вернет значение unit. Подобно этому, (hGetLine h) является действием, при выполнении которого считывается строка, вводимая с описателя h, которая возвращается в виде String. Используя имеющуюся в Haskell do-нотацию, мы можем объединить небольшие программы, имеющие побочные эффекты, чтобы получить более крупную программу, обладающую такими эффектами. Например, hEchoLine считывает строку из входных данных и отображает ее:

```
hEchoLine :: Handle -> IO String
hEchoLine h = do { s <- hGetLine h
                  : hPutStr h ("Только что считано: " ++ s ++ "\n")
                  : return s }
```

Запись do {a₁; ...; a_n} создает действие за счет объединения более мелких действий a₁...a_n в последовательность. Поэтому hEchoLine h — это действие, при выполнении которого сначала выполняется hGetLine h для считывания строки из h и присвоения результату имени s. Затем это действие выполнит hPutStr, чтобы вывести s, предварив² его значение строкой «Только что считано: ». И наконец, будет возвращена строка s. Это последняя строка интересна тем, что return не является встроенной конструкцией языка: это обычная функция, имеющая тип:

```
return :: a -> IO a
```

При выполнении действия return v возвращается v, не вызывая никаких побочных эффектов³. Эта функция воздействует на значения любого типа, и мы указываем на это, используя тип переменной a при указании типа самой функции.

Одной из важных разновидностей побочных эффектов является ввод-вывод. Другая разновидность относится к чтению или записи изменяемой переменной. Например, следующая функция увеличивает значение изменяемой переменной:

```
incRef :: IORef Int -> IO ()
incRef var = do { val <- readIORef var
                  : writeIORef var (val+1) }
```

В ней incRef var является действием, которое сначала выполняет readIORef var, считывая значение переменной, присваивает ее значение переменной val и затем выполняет writeIORef, чтобы записать значение (val+1) в переменную. readIORef и writeIORef имеют следующие типы:

```
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Значение типа IORef t должно рассматриваться как указатель или ссылка на изменяемое место, содержащее значение типа t, что немного похоже на тип (t *) в Си. В случае с incRef аргумент имеет тип IORef Int, поскольку incRef применяется только к местам, содержащим Int.

¹ Описатель (Handle) в Haskell играет ту же роль, что и файловый дескриптор в Си: он определяет, какой файл или канал использовать для чтения или записи. Как и в Unix, в Haskell есть три предопределенных описателя: stdin, stdout и stderr.

² Оператор ++ объединяет две строки.

³ Тип IO указывает на возможность, а не на неизбежность побочных эффектов.

До сих пор я объяснял, как создаются более крупные действия за счет объединения нескольких более мелких действий, но как действие приводится в исполнение? В Haskell вся программа определяется как одно IO-действие, назначенное `main`. Запуск программы является выполнением действия `main`. Вот как, к примеру, выглядит готовая программа:

```
main :: IO ()
main = do { hPutStr stdout "Hello"
           ; hPutStr stdout " world\n" }
```

Эта программа является последовательной, поскольку запись `do` объединяет IO-действия в последовательность. Для создания параллельной программы нам нужен еще один базисный элемент — `forkIO`:

```
forkIO :: IO a -> IO ThreadId
```

Встроенная в Haskell функция `forkIO` воспринимает в качестве своего аргумента IO-действие и вызывает его в виде параллельного потока Haskell. После своего создания это действие запускается параллельно со всеми другими потоками Haskell имеющейся в нем системой рабочего цикла. Представьте, к примеру, что мы внесли в программу `main` следующие изменения¹:

```
main :: IO ()
main = do { forkIO (hPutStr stdout "Hello")
           ; hPutStr stdout " world\n" }
```

Теперь два действия `hPutStr` будут запущены параллельно. Которое из них «победит» (первым осуществив вывод своей строки) — не указано. Потоки Haskell, порожденные `forkIO`, чрезвычайно легковесны: они занимают несколько сотен байт памяти, и для отдельной программы вполне приемлемо породить тысячи таких потоков.

У благосклонного читателя на данный момент может сложиться представление о том, что Haskell очень неуклюжий и многословный язык. В конце концов, наше трехстрочное определение `incRef` делает практически то же самое, что и `x++` на языке Си! Действительно, побочные эффекты в Haskell определены слишком явно и несколько многословно. Но во-первых, нужно помнить, что Haskell в первую очередь *функциональный* язык. Большинство программ написано в функциональном ядре Haskell, которое имеет довольно широкие возможности, является кратким и выразительным. Поэтому Haskell вдохновляет вас на написание программ с умеренным использованием побочных эффектов.

Во-вторых, заметьте, что явное определение побочных эффектов раскрывает немало ценной информации.

Рассмотрим две функции:

```
f :: Int -> Int
g :: Int -> IO Int
```

Всего лишь взглянув на их типы, мы можем понять, что `f` — это простая функция: она не вызывает побочных эффектов. Если присвоить `Int` конкретное значение, скажем 42, то вызов (`f 42`) всякий раз будет возвращать одно и то же

¹ В первой строке `main` мы могли бы вместо этого написать `tid <- forkIO (hPutStr ...)`, чтобы привязать значение `ThreadId`, возвращаемое `forkIO`, к `tid`. Но так как мы не используем возвращаемое значение `ThreadId`, то можем от него отказаться, опустив фрагмент `tid <-`.

значение. В отличие от этого, `g` имеет побочные эффекты, и это видно по ее типу. При каждом выполнении `g` она может возвращать разные результаты — к примеру, считывать информацию из `stdin` или модифицировать значение изменяемой переменной — даже если ее аргументы каждый раз остаются неизменными. В дальнейшем эта возможность обозначать побочные эффекты яным образом проявит свою несомненную пользу.

Наконец, действия являются значениями первого класса: они могут быть переданы в качестве аргументов, а также возвращены в качестве результатов. Вот как, к примеру, выглядит упрощенное определение функции цикла `for`, но не встроенное, а написанное целиком на языке Haskell:

```
nTimes :: Int -> IO () -> IO ()
nTimes 0 do_this = return ()
nTimes n do_this = do { do_this; nTimes (n-1) do_this }
```

Эта рекурсивная функция воспринимает переменную `Int`, показывающую, сколько раз нужно выполнить цикл и действие `do_this`; она возвращает действие, при выполнении которого `n` раз выполняется действие `do_this`. Вот как выглядит пример, использующий `nTimes` для десятикратного отображения `Hello`:

```
main = nTimes 10 (hPutStr stdout "Hello\n")
```

В сущности, рассматривая действия в качестве значений первого класса, Haskell поддерживает *управляющие структуры, определяемые пользователем*.

Эта глава не предназначена для полноценного введения в Haskell или в имеющиеся в нем побочные эффекты. Хорошей отправной точкой для дальнейшего чтения может послужить руководство «*Tackling the awkward squad*» («Занятия со звездом новобранцев»).

Транзакции в Haskell

Теперь мы можем вернуться к нашей функции перевода средств. Вот как выглядит ее код:

```
transfer :: Account -> Account -> Int -> IO ()
-- Перевод 'amount' со счета 'from' на счет 'to'
transfer from to amount
= atomically (do { deposit to amount
                  : withdraw from amount })
```

Теперь внутренний блок `do` говорит сам за себя: мы вызываем `deposit` для зачисления `amount` на счет `to` и `withdraw` для снятия `amount` со счета `from`. Эти дополнительные функции мы скоро напишем, но сначала рассмотрим вызов функции

¹ Simon Peyton Jones, «*Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*», edited by C. A. R. Hoare, M. Broy and R. Steinbrueggen, Engineering theories of software construction, Marktoberdorf Summer School 2000, NATO ASI Series, pp. 47–96. IOS Press, 2001.

`atomically`. Она воспринимает действие в качестве своего аргумента и выполняет его атомарно. Если точнее, то она гарантирует следующее.

Атомарность

Результаты атомарного действия становятся сразу же видимыми для всех других потоков. Тем самым гарантируется, что ни один другой поток не сможет наблюдать состояние, при котором деньги уже зачислены на счет то, но еще не сняты со счета `from`.

Изолированность

Во время вызова `atomically act` действие `act` совершенно не затрагивается другими потоками. Как будто при запуске `act` делается стоп-кадр состояния окружающего мира, а затем это действие выполняется над тем, что попало в этот кадр.

Простая модель выполнения `atomically` выглядит следующим образом. Предположим, есть единая глобальная блокировка. Тогда `atomically act` захватывает блокировку, выполняет действие `act` и снимает блокировку. Такая реализация дает твердые гарантии, что никакие два атомарных блока не могут быть выполнены одновременно, обеспечивая тем самым их атомарность.

У этой модели есть две проблемы. Во-первых, она не гарантирует полной изолированности: пока один поток получает доступ к `IORef` внутри атомарного блока (удерживая глобальную блокировку — *Global Lock*), ничто не мешает другому потоку вести запись в тот же `IORef` напрямую (то есть за пределами `atomically`, без удержания *Global Lock*), поэтому нарушение изолированности обеспечено. Во-вторых, из рук вон плохая производительность, поскольку каждый атомарный блок выстраивается в последовательность, даже если его работе фактически ничего не может помешать.

Вторую проблему нам еще предстоит рассмотреть в разделе «Реализация транзакционной памяти». А пока от первого недостатка можно легко избавиться с помощью системы типов. Мы присваиваем функции `atomically` следующий тип:

```
atomically :: STM a -> IO a
```

Аргументом `atomically` служит действие типа `STM a`. Действие `STM` похоже на действие `IO` тем, что оно может иметь побочные эффекты, но обладает при этом значительно меньшим диапазоном таких эффектов. Основное, что можно сделать в `STM`-действии, — это считать или записать значение транзакционной переменной, имеющей тип (`TVar a`), почти так же, как мы могли считать или записать значение `IORefs` в действии `IO`¹:

```
readTVar :: TVar a -> STM a  
writeTVar :: TVar a -> a -> STM ()
```

¹ Здесь использована непоследовательная спецификация: было бы логичнее воспользоваться либо `TVar` и `IOVar`, либо `TRef` и `IORef`. Но на данном этапе это изменение нарушило бы ход повествования; так что, хорошо это или плохо, но у нас есть `TVar` и `IORef`.

STM-действия могут быть объединены все той же записью do, как и IO-действия — запись do перегружается для работы с обоими типами, как это происходит и с return¹. Вот как, к примеру, выглядит код для withdraw:

```
type Account = TVar Int

withdraw :: Account -> Int -> STM ()
withdraw acc amount
  = do { bal <- readTVar acc
        : writeTVar acc (bal - amount) }
```

Мы представляем Account транзакционной переменной, содержащей Int для баланса счета. Затем STM-действие withdraw, которое уменьшает баланс счета на величину amount.

Для завершения определения transfer мы можем определить deposit на языке withdraw:

```
deposit :: Account -> Int -> STM ()
deposit acc amount = withdraw acc (- amount)
```

Заметьте, что в конечном счете transfer выполняет четыре элементарных действия чтения-записи: считывание, а затем запись на счет to сопровождается считыванием, а затем записью на счет from. Эти четыре действия выполняются автоматически, и это соответствует спецификации, заданной в начале раздела «Простой пример: банковские счета».

Система типов явным образом уберегает нас от чтения или записи TVar за пределами транзакции. Представим, к примеру, что мы попытались сделать следующее:

```
bad :: Account -> IO ()
bad acc = do { hPutStr stdout "Снятие средств..."
               : withdraw acc 10 }
```

Такая программа неработоспособна, поскольку hPutStr — это IO-действие, а withdraw — STM-действие, и эти два действия не могут объединяться в едином do-блоке. Если мы заключим withdraw в функцию atomically, то все будет в порядке:

```
good :: Account -> IO ()
good acc = do { hPutStr stdout "Снятие средств..."
               : atomically (withdraw acc 10) }
```

¹ Такая перегрузка записи do и return не является каким-то специализированным приемом для поддержки IO и STM. Скорее, и IO и STM являются примерами общего шаблона, названного монадой (описан П. Уэдлером (P. L. Wadler) в «The essence of functional programming», 20th ACM Symposium on Principles of Programming Languages [POPL '92], Альбукерк, pp. 1–14, ACM, Jan. 1992), а перегрузка достигается за счет представления, что общий шаблон пользуется тем же самым общим механизмом type-class (описан П. Уэдлером (P. L. Wadler) и С. Блоттом (S. Blott) в «How to make ad-hoc polymorphism less ad hoc», Proc 16th ACM Symposium on Principles of Programming Languages, Остин, Техас, ACM, Jan. 1989; и Саймоном Пейтоном Джонсом (Simon Peyton Jones), Марком Джонсом (Mark Jones) и Эриком Мейджером (Erik Meijer) в «Type classes: an exploration of the design space», Haskell workshop, Amsterdam, 1997).

Реализация транзакционной памяти

Рассмотренные ранее гарантии атомарности и изолированности — это все, что нужно программисту для использования STM. Но памятую об этом, я следовал своей интуиции и многократно убеждался в пользе разумной модели реализации, и одну такую модель я собираюсь коротко рассмотреть в этом разделе. Следует помнить, что это лишь *один* из возможных вариантов реализации. Одним из достоинств STM-абстракции является предоставление небольшого, но вполне понятного интерфейса, который может быть реализован множеством способов, как простых, так и сложных.

Одна особенно впечатляющая реализация хорошо зарекомендовала себя в мире баз данных, я имею в виду *оптимистическое выполнение*. При выполнении `atomically act` резервируется изначально пустой *транзакционный журнал*, относящийся к локальному потоку. Затем выполняется действие `act`, которое вообще не устанавливает никаких блокировок. Во время выполнения `act` каждый вызов `writeTVar` записывает адрес `TVar` и его новое значение в журнал; запись в саму переменную `TVar` не ведется. При каждом вызове `readTVar` сначала ведется поиск в журнале (на тот случай, если в нем есть запись о `TVar`, сделанная более ранним вызовом `writeTVar`); если такая запись не будет найдена, значение считывается из самой `TVar`, и `TVar` вместе со считанным значением записывается в журнал. Тем временем другие потоки могут запускать свои собственные атомарные блоки, считывая и записывая переменные `TVar` сколько им угодно.

Когда действие `act` завершится, реализация сначала *проверяет* журнал, и если проверка будет успешной, *фиксирует* его. На этапе проверки изучается каждая записанная в журнал переменная `readTVar`, и контролируется соответствие значения в журнале текущему значению реальной переменной `TVar`. Если значения соответствуют, проверка считается успешной, и на этапе фиксации берутся все записи, внесенные в журнал, и записываются в реальные переменные `TVar`.

Этапы выполняются неразрывно друг от друга: реализация отключает прерывания или использует блокировки или команды сравнения и перестановки, то есть все, что необходимо для гарантии восприятия абсолютной неразрывности проверки и фиксации со стороны других потоков. Но все это управляет самой реализацией, и программисту не нужно знать или беспокоиться о том, как это делается.

А если проверка потерпит неудачу? Значит, у транзакции было непоследовательное представление памяти. Поэтому мы прекращаем транзакцию, заново инициализируем журнал и перезапускаем действие. Этот процесс называется *повторным выполнением*. Поскольку никакие записи действий `act` не были зафиксированы в памяти, повторный запуск совершению безопасен. Тем не менее следует заметить важность отсутствия в действиях `act` других эффектов, кроме считывания и записи значений переменных `TVar`. Рассмотрим, к примеру, следующий код:

```
atomically (do { x <- readTVar xv  
                  : y <- readTVar yv
```

```
: if x>y then launchMissiles
    else return () {}}
```

где `launchMissiles :: IO ()` приводит к серьезным международным побочным эффектам¹. Поскольку атомарные блоки выполняются без использования блокировок, то это может привести к непоследовательному представлению памяти в том случае, если другие потоки параллельно модифицируют `xv` и `uv`. Случись такое, это может привести к ошибочному пуску ракет, и лишь затем выяснится, что проверка не прошла и транзакция должна быть перезапущена. К счастью, система типов оберегает нас от запуска 10-действий внутри STM-действий, поэтому приведенный выше фрагмент будет отвергнут проверкой типов. В этом заключается еще одно большое преимущество различий типов `IO`- и `STM`-действий.

Блокировка и выбор

Атомарные блоки в соответствии со сложившимся о них на данный момент представлением совершенно не подходят для согласования параллельных программ. В них нет двух ключевых средств: *блокировки* и *выбора*. В этом разделе я расскажу, как разработан основной STM-интерфейс, чтобы включить их абсолютно модульным путем.

Предположим, что поток должен *блокироваться*, если попытается превысить кредит счета (то есть снять с него сумму, превышающую текущий баланс). Подобные ситуации — не редкость в параллельных программах: к примеру, поток должен блокироваться при считывании из пустого буфера или при ожидании наступления какого-нибудь события. В STM мы достигаем этого за счет добавления отдельной функции `retry`, имеющей следующий тип:

```
retry :: STM a
```

А вот как выглядит измененная версия `withdraw` с блокировкой при переходе баланса в отрицательное состояние:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
  = do { bal <- readTVar acc
        : if amount > 0 && amount > bal
            then retry
            else writeTVar acc (bal - amount) }
```

Семантика `retry` довольно проста: если выполняется действие `retry`, то текущая транзакция отвергается, и попытка ее повторного проведения предпринимается чуть позже. Вполне приемлемо сразу же повторить транзакцию, но эта попытка вряд ли будет удачной: скорее всего, состояние счета еще не изменится, и транзакция снова приведет к вызову `retry`. Эффективная реализация вместо этого заблокирует поток до тех пор, пока какой-нибудь другой поток не осуществит запись в `acc`. Но откуда реализация узнает, что нужно ждать записи в `acc`?

¹ Launch missiles — пуск ракет. — Примеч. пер.

Это произойдет благодаря тому, что транзакция на пути к `retry` считывает значение `acc`, и этот факт просто записывается в транзакционный журнал.

Условное выражение `limitedWithdraw` имеет весьма распространенную схему: проверить булево условие, и если оно не удовлетворено, вызвать `retry`. Этую схему очень просто абстрагировать в виде функции `check`:

```
check :: Bool -> STM ()
check True = return ()
check False = retry
```

Теперь мы можем воспользоваться `check`, чтобы уточнить `limitedWithdraw`:

```
limitedWithdraw :: Account -> Int -> STM ()
limitedWithdraw acc amount
= do { bal <- readTVar acc
      : check (amount <= 0 || amount <= bal)
      : writeTVar acc (bal - amount) }
```

Теперь переключим наше внимание на *выбор*. Предположим, что нужно снять средства со счета А, если их там достаточно, но если их не хватает, то снять средства со счета Б. Для этого нам нужно иметь возможность выбора альтернативного действия, если первое действие отправляется на повторное выполнение. Для поддержки выбора STM Haskell содержит еще одно элементарное действие под названием `orElse`, которое имеет следующий тип:

```
orElse :: STM a -> STM a -> STM a
```

Как и `atomically`, `orElse` воспринимает действия в качестве своих аргументов и объединяет их вместе в одно большое действие. Оно обладает следующей семантикой: действие (`orElse a1 a2`) сначала выполняет `a1`; если `a1` отправляется на повторное выполнение (то есть вызывает `retry`), оно вместо этого пробует выполнить `a2`. Если `a2` также отправляется на повторное выполнение, то повторно выполняется все действие. Проще, наверное, посмотреть на использование действия `orElse`:

```
limitedWithdraw2 :: Account -> Int -> STM ()
-- (limitedWithdraw2 acc1 acc2 amt) снимает amt со счета acc1.
-- если на acc1 достаточно средств.
-- если нет, средства снимаются со счета acc2.
-- Если ни один счет не имеет достаточных средств, повторно выполняется все
-- действие.
limitedWithdraw2 acc1 acc2 amt
= orElse (limitedWithdraw acc1 amt) (limitedWithdraw acc2 amt)
```

Поскольку результат `orElse` сам по себе является STM-действием, вы можете предложить его другому вызову `orElse` и выбирать, таким образом, из произвольного числа альтернативных действий.

Краткая сводка основных STM-операций

В этом разделе будут представлены все ключевые операции транзакционной памяти, поддерживаемые STM Haskell. Они сведены в табл. 24.1. В нее включена и одна, ранее не встречавшаяся нам операция: `newTVar`, предоставляющая

способ создания новых элементов `TVar`, которым мы воспользуемся в следующем разделе.

Таблица 24.1. Ключевые операции STM Haskell

Операция	Сигнатура типа
<code>Atomically</code>	<code>STM a -> IO a</code>
<code>Retry</code>	<code>STM a</code>
<code>orElse</code>	<code>STM a -> STM a -> STM a</code>
<code>newTVar</code>	<code>a -> STM (TVar a)</code>
<code>readTVar</code>	<code>TVar a -> STM a</code>
<code>writeTVar</code>	<code>TVar a -> a -> STM ()</code>

Задача Санта-Клауса

Я хочу показать вам полноценную, работоспособную параллельную программу, использующую STM. Возьмем хорошо известную так называемую задачу Санта-Клауса¹, изначально приписываемую Троно (Trono)².

Санта периодически спит, пока не будет разбужен либо всеми своими девятью северными оленями, вернувшимися со свободной выпаски, либо группой из трех эльфов, которых у него всего девять. Если его разбудят олени, он запрягает каждого из них в сани, доставляет вместе с ними игрушки, и в заключение распрягает их (отпуская на свободную выпаску). Если его разбудят эльфы, он ведет каждую группу в свой кабинет, совещается с ними по поводу разработки новых игрушек, а в заключение выводит каждого из них из кабинета (давая возможность вернуться к работе). Если Санта-Клауса будут одновременно ждать и группа эльфов и группа оленей, он отдаст приоритет оленям.

Использование хорошо известного примера позволит вам провести непосредственное сравнение моего решения с подробно описанными решениями на других языках программирования. В частности, в статье Троно (Trono) приводится частично верное решение, основанное на применении семафоров. Бен-Ари (Ben-Ari) приводит решение на Ada95 и на Ada³. Бентон (Benton) приводит решение на Polyphonic C#⁴.

¹ Мой выбор был навеян тем, что я писал эти слова 22 декабря.

² J. A. Trono, «A new exercise in concurrency», SIGCSE Bulletin, vol. 26, pp. 8–10, 1994.

³ Mordechai Ben-Ari, «How to solve the Santa Claus problem», Concurrency: Practice and Experience, vol. 10, № 6, pp. 485–496, 1998.

⁴ Nick Benton, «Jingle bells: Solving the Santa Claus problem in Polyphonic C#», Technical report, Microsoft Research, 2003.

Олени и эльфы

Основная идея реализации на STM Haskell заключается в следующем. Санта составляет одну группу (Group) для эльфов и одну – для оленей. Каждый эльф (или олень) старается присоединиться к своей группе – Group. Если ему это удастся, то он, в свою очередь, получает двое ворот (Gates). Первые ворота – Gate, позволяют Санте контролировать вход эльфа в кабинет, а также дают ему знать, когда все эльфы оказываются в кабинете. Аналогично этому, вторые ворота позволяют контролировать выход эльфов из кабинета. Со своей стороны, Санта ждет готовности обеих своих двух групп, а затем использует ворота групп – Group Gate для выстраивания своих помощников (эльфов или оленей) и для проведения их по соответствующим задачам. Таким образом, помощники проводят свои жизни в бесконечном цикле: стараются присоединиться к группе, проходят ворота, контролируемые Санта-Клаусом, а затем выжидают неопределенное время перед новой попыткой присоединения к группе.

Переложение этого свободного описания на язык Haskell дает применительное к эльфам следующий код¹:

```
elf1 :: Group -> IO ()  
elf1 group elf_id = do { (in_gate, out_gate) <- joinGroup group  
    : passGate in_gate  
    : meetInStudy elf_id  
    : passGate out_gate }
```

Эльфу – elf передается его группа – Group и число Int, определяющее личность эльфа. Эта идентификация используется только в вызове функции meetInStudy, которая просто выводит сообщение о происходящем²:

```
meetInStudy :: Int -> IO ()  
meetInStudy id = putStrLn ("Эльф " ++ show id ++ " совещается в кабинете\n")
```

Эльф вызывает joinGroup, чтобы присоединиться к своей группе, и passGate, чтобы пройти через каждые из ворот:

```
joinGroup :: Group -> IO (Gate, Gate)  
passGate :: Gate -> IO ()
```

Для оленя используется тот же код, за исключением того что олень доставляет игрушки, а не совещается в кабинете:

```
deliverToys :: Int -> IO ()  
deliverToys id = putStrLn ("Олень " ++ show id ++ " доставляет игрушки\n")
```

Поскольку IO-действия относятся к первому классу, мы можем кратко выразить все это в следующем общем шаблоне:

```
helper1 :: Group -> IO () -> IO ()  
helper1 group do_task = do { (in_gate, out_gate) <- joinGroup group  
    : passGate in_gate
```

¹ Я присвоил этой функции суффикс 1, поскольку она работает только с одним жизненным циклом эльфа, а в действительности эльфы, справившись со своей задачей, снова присоединяются к забавам. Мы займемся определением эльфов в разделе «Основная программа».

² PutStr – библиотечная функция, вызывающая hPutStr stdout.

```
; do_task
; passGate out_gate }
```

Второй аргумент helper1 является IO-действием, являющимся задачей, выполняемой помощником между двумя вызовами прохода ворот — passGate. Теперь мы можем приспособить helper1, чтобы он был и эльфом, и оленем:

```
elf1. reindeer1 :: Group -> Int -> IO ()
elf1      gp id = helper1 gp (meetInStudy id)
reindeer1 gp id = helper1 gp (deliverToys id)
```

Ворота и группы

Первой абстракцией будут ворота — Gate, поддерживающие следующий интерфейс:

```
newGate :: Int -> STM Gate
passGate :: Gate -> IO ()
operateGate :: Gate -> IO ()
```

У Gate есть фиксированная *пропускная способность* n, определяемая при создании новых ворот и изменяемая величина *оставшихся пропусков*. Эта величина уменьшается при каждом вызове помощником passGate для того, чтобы пройти через ворота; если число оставшихся пропусков равно нулю, passGate блокируется. Gate создаются с нулевым количеством оставшихся пропусков, поэтому помощники пройти сквозь них не могут. Санта открывает ворота с помощью функции operateGate, которая опять устанавливает количество оставшихся пропусков в n.

Вот как выглядит возможная реализация ворот:

```
data Gate = MkGate Int (TVar Int)

newGate :: Int -> STM Gate
newGate n = do { tv <- newTVar 0: return (MkGate n tv) }

passGate :: Gate -> IO ()
passGate (MkGate n tv)
  = atomically (do { n_left <- readTVar tv
                      : check (n_left > 0)
                      : writeTVar tv (n_left-1) })

operateGate :: Gate -> IO ()
operateGate (MkGate n tv)
  = do { atomically (writeTVar tv n)
        : atomically (do { n_left <- readTVar tv
                           : check (n_left == 0) }) }
```

В первой строке Gate объявляются новым *типом данных*, с отдельным *конструктором данных* — MkGate. Конструктор имеет два поля: Int, которое передает пропускную способность ворот, и Tvar, чье содержимое сообщает, сколько помощников может пройти сквозь ворота, пока они не закроются. Если TVar содержит нуль, ворота закрыты.

Функция `newGate` создает новые ворота, выделяя пространство памяти под `TVar` и создавая значение `Gate` за счет вызова конструктора `MkGate`. Функция `passGate` выполняет двойную работу: использует сопоставление с шаблоном, чтобы разобрать конструктор `MkGate`, а затем уменьшает содержимое `TVar`, проводя проверку, чтобы убедиться в остатке пропускной способности ворот, точно так же, как мы делали это с `withdraw` в разделе «Блокировка и выбор». В заключение `operateGate` сначала открывает ворота, записывая их полную пропускную способность в `TVar`, а затем ждет, пока значение `TVar` не будет уменьшено до нуля.

Группа — `Group` имеет следующий интерфейс:

```
newGroup :: Int -> IO Group
joinGroup :: Group -> IO (Gate.Gate)
awaitGroup :: Group -> STM (Gate.Gate)
```

Кроме того, группа — `Group` создается пустой, с определенной емкостью. А эльф может присоединиться к группе, вызвав функцию `joinGroup`, но этот вызов блокируется, если группа уже заполнена. Санта вызывает функцию `awaitGroup`, чтобы дождаться заполнения группы; когда группа заполнена, он устанавливает для нее ворота, и `Group` тут же устанавливается в новое исходное состояние с новыми воротами, чтобы другая группа желающих эльфов могла приступить к объединению.

Возможная реализация выглядит следующим образом:

```
data Group = MkGroup Int (TVar (Int, Gate, Gate))
newGroup n = atomically (do { g1 <- newGate n; g2 <- newGate n
    : tv <- newTVar (n, g1, g2)
    : return (MkGroup n tv) })
```

Тут `Group` также объявляется новым типом данных, с конструктором `MkGroup` и двумя полями: полной емкостью группы, и `TVar`, в котором содержится число вакансий, а также с двумя воротами группы. Создание новой группы является поводом для создания новых ворот, инициализации нового значения `TVar` и возврата структуры, построенной с помощью `MkGroup`.

Теперь реализация `joinGroup` и `awaitGroup` более-менее определена следующими структурами данных:

```
joinGroup (MkGroup n tv)
= atomically (do { (n_left, g1, g2) <- readTVar tv
    : check (n_left > 0)
    : writeTVar tv (n_left-1, g1, g2)
    : return (g1,g2) })
awaitGroup (MkGroup n tv)
= do { (n_left, g1, g2) <- readTVar tv
    : check (n_left == 0)
    : new_g1 <- newGate n; new_g2 <- newGate n
    : writeTVar tv (n,new_g1,new_g2)
    : return (g1,g2) }
```

Учтите, что `awaitGroup` создает новые ворота при проведении новой инициализации `Group`. Это гарантирует возможность сбора новой группы, в то время

как предыдущая группа все еще беседует с Сантой в его кабинете, без опаски, что эльф из новой группы обгонит сонного эльфа из старой.

Просматривая этот раздел, вы могли заметить, что я присвоил некоторым операциям Group и Gate типы I0 (например newGroup, joinGroup), а некоторым – типы STM (например newGate, awaitGroup).

Как я выбирал, какой именно тип присвоить? Например, newGroup имеет тип I0, что означает, что я никогда не смогу ее вызывать из STM-действия. Но это лишь вопрос удобства: вместо этого я мог бы присвоить newGroup тип STM, опустив в ее определении функцию atomically. В обмен на это я должен был бы написать atomically (newGroup n) в каждом месте вызова, а не просто newGroup n. Преимущество присвоения newGate типа STM состоит в большей пригодности к компоновке, если newGroup в этой программе не понадобится. В другой ситуации я хотел бы вызвать newGate из newGroup и поэтому присвоил newGate тип STM.

Вообще-то при разработке библиотеки нужно присваивать функциям тип STM везде, где только это возможно. STM-действия можно рассматривать как своеобразные лего-элементы, которые могут быть собраны воедино, с использованием do {...}, retry иorElse, чтобы создать более крупные STM-действия. Но как только вы поместите блок в функцию atomically, сделав его I0-типов, он уже не сможет быть атомарно объединен с другими действиями. Для этого есть вполне серьезные основания: значение I0-типа может выполняться произвольно, необратимо осуществляя ввод–вывод (такой как запуск ракет – launch Missiles).

Поэтому хорошей библиотечной конструкцией считается экспорт STM-действий (а не I0-действий) везде, где это возможно, поскольку они легко компонуются; их тип извещает, что они не имеют необратимых эффектов. Пользователь библиотеки легко перейдет от STM к I0 (используя atomically), но никак не наоборот.

Но иногда использование I0-действия просто *необходимо*. Взглянем на operateGate. Два вызова atomically не могут быть объединены в один, поскольку первый имеет видимый за его пределами побочный эффект (открытие ворот), в то время как второй блок ждет, пока все эльфы не проснутся и не пройдут сквозь ворота. Поэтому operateGate *должен* иметь тип I0.

Основная программа

Сначала нам нужно реализовать внешнюю структуру программы, хотя мы еще не реализовали самого Санта-Клауса. Она выглядит следующим образом:

```
main = do { elf_group <- newGroup 3
           : sequence_ [ elf elf_group n | n <- [1..10] ]
           : rein_group <- newGroup 9
           : sequence_ [ reindeer rein_group n | n <- [1..9] ]
           : forever (santa elf_group rein_group) }
```

В первой строке создается Group для эльфов, емкостью 3. Вторая строка более затейлива: в ней используется так называемое списочное включение для соз-

дания списка IO-действий, и вызывается `sequence_` для их последовательного выполнения. Списочное включение `[e|xs]` читается как «список всех `e`, где `x` берется из списка `xs`». Поэтому аргументом `sequence_` является список:

```
[elf elf_group 1, elf elf_group 2, ..., elf elf_group 10]
```

Каждый из этих вызовов приводит к IO-действию, которое порождает поток эльфа. Функция `sequence_` воспринимает список IO-действий и возвращает действие, которое при выполнении запускает по порядку каждое действие списка¹:

```
sequence_ :: [IO a] -> IO ()
```

Функция `elf` создана на основе `elf1`, но имеет два отличия. Во-первых, нам нужно, чтобы эльф жил по бесконечному циклу, и во-вторых, нам нужно, чтобы он запускался в отдельном потоке:

```
elf :: Group -> Int -> IO ThreadId
elf gp id = forkIO (forever (do { elf1 gp id; randomDelay }))
```

Элемент `forkIO` на основе своих аргументов порождает отдельный Haskell-поток (см. предыдущий раздел «Побочные эффекты и ввод–вывод в языке Haskell»). В свою очередь, аргумент `forkIO` является вызовом функции `forever`, которая многократно запускает свои аргументы (сравните с определением `nTimes` в разделе «Побочные эффекты и ввод–вывод в языке Haskell»):

```
forever :: IO () -> IO ()
-- Многократное выполнение действия
forever act = do { act; forever act }
```

В конечном счете выражение `(elf1 gp id)` является IO-действием, и нам нужно повторять это действие многократно, всякий раз сопровождая его произвольной задержкой:

```
randomDelay :: IO ()
-- Произвольная задержка продолжительностью от 1 до 1000000 микросекунд
randomDelay = do { waitTime <- getStdRandom (randomR (1, 1000000))
                  : threadDelay waitTime }
```

Остальная часть основной программы должна быть понятна сама по себе. Мы создаем девятерых оленей тем же способом, которым создавали десятерых эльфов, и исключение состоит в том, что мы вызываем `reindeer` вместо `elf`:

```
reindeer :: Group -> Int -> IO ThreadId
reindeer gp id = forkIO (forever (do { reindeer1 gp id; randomDelay }))
```

Код основной программы — `main` завершается повторным использованием `forever` для многократного запуска `santa`. Остается только реализовать работу самого Санта-Клауса.

Реализация Санта-Клауса

Санта-Клаус — самый интересный участник этой небольшой драмы, поскольку выбор всегда за ним. Он должен ждать, пока не соберется либо группа ожидаю-

¹ Тип `[IO a]` означает «список значений типа `IO a`». Знак подчеркивания в имени `sequence_` может вызвать удивление: его присутствие обусловлено родством с функцией `sequence`, имеющей тип `[IO a] -> IO [a]`, которая собирает в список результаты действий, переданных в качестве аргумента. Обе функции, `sequence` и `sequence_`, определены в библиотеке `Prelude`, импортируемой по умолчанию.

ших оленей, либо группа эльфов. Как только он выбрал, какой группе уделить внимание, он должен провести их по соответствующим задачам. Его код выглядит следующим образом:

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
= do { putStrLn "-----\n"
      : (task, (in_gate, out_gate))
      <- atomically (orElse
                      (chooseGroup rein_gp "доставлять игрушки")
                      (chooseGroup elf_gp "совещаться в кабинете"))
      : putStrLn ("Эй! Эй! Эй! давайте " ++ task ++ "\n")
      : operateGate in_gate
      -- Теперь помощники выполняют свои задачи
      : operateGate out_gate }
```

where

```
chooseGroup :: Group -> String -> STM (String, (Gate.Gate))
chooseGroup gp task = do { gates <- awaitGroup gp
                           ; return (task, gates) }
```

Выбор делается с помощью функции `orElse`, которая сначала пытается выбрать оленей (предоставляя им, таким образом, приоритет) и только при других обстоятельствах выбирает эльфов. Функция `chooseGroup` осуществляет вызов соответствующей группы с помощью `awaitGroup` и возвращает пару, состоящую из строки с признаком задачи (доставлять игрушки или совещаться в кабинете) и двое ворот, которыми Санта может управлять, проводя группу по задаче. Как только будет сделан выбор, Санта выводит сообщение и последовательно управляет двумя воротами.

Эта реализация вполне работоспособна, но мы еще рассмотрим альтернативную, более общую версию, поскольку `santa` иллюстрирует довольно распространенную схему программирования. Схема состоит в следующем: поток (в данном случае Санта) выбирает одну из атомарных транзакций, за которой следуют одна или более дальнейших последовательных транзакций. Можно привести и другой типичный пример: получение сообщения из одной или нескольких очередей сообщений, обработка сообщения и повторение цикла. В сценарии с участием Санта-Клауса последовательное действие было очень похожим и для эльфов и для оленей — в обоих случаях Санта должен вывести сообщение и управлять двумя воротами. Но это не сработает, если Санта-Клаусу придется заниматься для эльфов и оленей каким-то существенно отличающимся делом. Одним из подходов в решении этой задачи может стать возвращение булева значения, свидетельствующего о том, что именно было выбрано, и диспетчеризация на основе этого булева значения после выбора, но такой подход становится неудобным при добавлении других альтернативных вариантов. Следующий подход работает лучше:

```
santa :: Group -> Group -> IO ()
santa elf_gp rein_gp
= do { putStrLn "-----\n"
      : choose [(awaitGroup rein_gp, run "доставлять игрушки"),
                  (awaitGroup elf_gp, run "совещаться в кабинете")]
      : run }
```

where

```
run :: String -> (Gate.Gate) -> IO ()
```

```

run task (in_gate.out_gate)
  = do { putStrLn ("Эй! Эй! Эй! Давайте " ++ task ++ "\n")
        : operateGate in_gate
        : operateGate out_gate }

```

Функция choose подобна отслеживающей команде: она воспринимает список пар, ждет, пока первый компонент пары будет готов «выстрелить», а затем выполняет второй компонент. Итак, choose имеет следующий тип¹:

```
choose :: [(STM a, a -> IO ())] -> IO ()
```

В качестве отслеживателя (guard) выступает STM-действие, выдающее значение типа a; когда STM-действие готово (то есть оно не отправляется на повторное выполнение), choose может передать значение второму компоненту, который по этой причине должен быть функцией, ожидающей значение типа a. Памятую об этом, можно легко разобраться в функции santa. Она использует awaitGroup для ожидания готовности группы; функция choose получает двое ворот, возвращенных awaitGroup, и передает их функции g1p. Последняя последовательно управляет двумя воротами, повторно вызывая их блоки operateGate до тех пор, пока все эльфы (или олени) не пройдут сквозь ворота.

Код у функции choose короткий, но несколько замысловатый:

```

choose :: [(STM a, a -> IO ())] -> IO ()
choose choices = do { act <- atomically (foldrl orElse actions)
                     : act }
  where
    actions :: [STM (IO ())]
    actions = [ do { val <- guard; return (rhs val) }
               | (guard, rhs) <- choices ]

```

Сначала он формирует список actions, состоящий из STM-действий, которые затем объединяются при помощи orElse. (Вызов foldrl ? [x₁...x_n] возвращает x₁ ? x₂ ? ... ? x_n.) Каждое из этих STM-действий само возвращает IO-действие, а именно то, что должно быть выполнено, когда будет сделан выбор. Именно поэтому каждое действие в списке имеет замысловатый тип STM (IO ()). Код сначала делает атомарный выбор среди списка альтернатив, получая взамен действие act с типом IO (), а затем выполняет действие act. Список actions выстраивается за счет извлечения каждой пары (guard, rhs) из списка вариантов (choices), запуска guard (STM-действия) и возвращения IO-действия, полученного за счет применения rhs к значению, возвращенному guard.

Компиляция и запуск программы

Я представил *весь* код этого примера. Если в его начало еще добавить соответствующие операторы импорта, показанные далее, то с программой можно будет поработать²:

¹ В Haskell тип [ty] означает список, чьи элементы имеют тип ty. В данном случае аргументы choose — это список пар, записанный в виде (ty₁.ty₂); первый компонент пары имеет тип STM a, а второй — является функцией, имеющей тип a->IO () .

² Код можно получить по адресу: <http://research.microsoft.com/~simonpj/papers/stm/Santa.hs.gz>.

```
module Main where
  import Control.Concurrent.STM
  import Control.Concurrent
  import System.Random
```

Для компилирования кода воспользуйтесь Glasgow Haskell Compiler, GHC¹:

```
$ ghc Santa.hs -package stm -o santa
```

И в завершение программу можно запустить на выполнение:

```
$ ./santa
Эй! Эй! Эй! Давайте доставлять игрушки
Олень 9 доставляет игрушки
Олень 8 доставляет игрушки
Олень 7 доставляет игрушки
Олень 6 доставляет игрушки
Олень 5 доставляет игрушки
Олень 4 доставляет игрушки
Олень 3 доставляет игрушки
Олень 2 доставляет игрушки
Олень 1 доставляет игрушки
Эй! Эй! Эй! Давайте совещаться в кабинете
Эльф 3 совещается в кабинете
Эльф 2 совещается в кабинете
Эльф 1 совещается в кабинете
... и т. д.
```

Размышления о языке Haskell

Haskell в первую очередь является функциональным языком. Несмотря на это, я полагаю, что он также и самый красивый в мире императивный язык. Если рассматривать Haskell в качестве императивного языка, то в нем обнаруживаются следующие необычные свойства:

- действия (имеющие эффекты) за счет использования системы типов четко отличаются от простых значений;
- действия являются значениями первого класса. Они могут передаваться функциям, возвращаться в качестве результатов, выстраиваться в списки и т. д., и все это не вызывает никаких побочных эффектов.

Используя действия в качестве значений первого класса, программист может определять *специализированные управляющие структуры*, вместо того чтобы обходиться структурами, предоставленными конструкторами языка. К примеру, `nTimes` – это простой цикл `for`, а `choose` реализует своеобразную команду отслеживания. Мы были свидетелями и других применений действий в качестве значений. В основной программе мы использовали имеющийся в Haskell богатый язык выражений (в данном случае списочные включения) для генерации списка действий, которые затем выполнялись по порядку благодаря использованию функции `sequence_`. Чуть раньше, при определении `helper1`, мы улучшили

¹ GHC свободно выложен по адресу: <http://haskell.org/ghc>.

модульность, убрав действие за пределы фрагмента кода. Чтобы проиллюстрировать это положение, я, возможно, злоупотребил в коде Санта-Клауса, являющимся весьма скромной программой, мощностью имеющихся в Haskell абстракций. Но для более крупных программ важность представления действий в качестве значений трудно переоценить.

С другой стороны, сосредоточившись на параллельной работе, я не раскрыл других аспектов языка Haskell – функций высокого порядка, отложенных вычислений, типов данных, полиморфизма, классов типов и т. д. Не всякая программа на Haskell обладает такой же императивностью, как наша программа! Большой объем информации о языке Haskell, включая книги, руководства, компиляторы и интерпретаторы Haskell, его библиотеки, рассылки и многое другое, можно найти по адресу <http://haskell.org>.

Вывод

Моя главная цель состоит в том, чтобы убедить вас, что используя STM, можно создавать программы, обладающие существенно большими свойствами модульности, чем при использовании блокировок и переменных условий. Прежде всего, следует отметить, что транзакционная память позволяет полностью избежать многих типичных проблем, являющихся настоящим бедствием для параллельных программ, основанных на использовании блокировок (я объяснял это в разделе «Порочность блокировок»). *Ни одна из этих проблем в STM Haskell не возникает.* Система типов оберегает вас от чтения или записи TVar за пределами атомарного блока, а при *отсутствии* программно-видимых блокировок вопросы о том, какие блокировки применить и в какой последовательности, просто не возникают. Во-вторых, еще одно преимущество STM, для которого здесь не хватило места, включает избавление от упущеных активизаций, обработки исключений и восстановления после ошибок.

В разделе «Порочность блокировок» мы также выяснили, что самая неприятная проблема программирования на основе блокировок состоит в том, что *блокировки не поддаются компоновке*. В отличие от этого, любая функция Haskell, имеющая тип STM, может быть скомпонована благодаря использованию последовательностей или выборов, с любой другой функцией STM-типа, чтобы создать новую функцию STM-типа. Кроме того, составная функция дает гарантию всех тех же атомарных свойств, что и отдельная функция. В частности, блокировка (retry) и выбор (orElse), которые по своей сути не обладают модульностью при выражении с использованием блокировок, могут быть полностью модульными в STM. Рассмотрим, к примеру, следующую транзакцию, в которой используются функции, определенные нами в разделе «Блокировка и выбор»:

```
atomically (do { limitedWithdraw a1 10
                  : limitedWithdraw2 a2 a3 20 })
```

Эта транзакция блокируется до тех пор, пока a1 не будет содержать как минимум 10 единиц, а a2 либо a3 не будут содержать 20 единиц. Но эти сложные условия блокировки не написаны в явном виде программистом, и несомненно, если функции limitedWithdraw реализованы в какой-то сложной библиотеке,

программист может даже не догадываться, как в них выстраиваются условия блокировки. STM обладает модульностью: небольшие программы могут быть объединены в более крупные программы *без раскрытия их внутренней реализации*.

У транзакционной памяти еще множество аспектов, не охваченных в этом кратком обзоре, включая такие важные темы, как вложенные транзакции, исключение, успех, «зависание» и инварианты. Описания многих из этих аспектов можно найти в статьях об STM Haskell¹.

Транзакционная память особенно хорошо «вписывается» в язык Haskell. В STM реализация потенциально должна отслеживать каждую загрузку памяти и сохранение данных, но Haskell STM нужны лишь *TVar*-операции, и они составляют весьма незначительную долю всех загрузок памяти и сохранений, выполняемых программой на Haskell. Кроме того, обработка действий как значений первого класса и развитая система типов позволяют нам предлагать твердые гарантии отсутствия помех без каких-либо расширений языка. Но ничего не содержит внедрение транзакционной памяти в основные императивные языки, хотя это внедрение может оказаться менее элегантным и потребовать большей языковой поддержки. Разумеется, реализация этой задачи является актуальной темой исследований; соответствующий всесторонний обзор представлен Ларусом (*Larus*) и Раджваром (*Rajwar*)².

Использование STM похоже на применение языка высокого уровня вместо ассемблерного кода — можно, конечно, и здесь допустить программные упущения, но для многих коварных ошибок просто нет повода, что в значительной мере способствует концентрации внимания на высокоуровневых аспектах программы. Но увы, «серебряной пули», которая облегчила бы написание параллельных программ, не существует. Тем не менее STM представляется многообещающим шагом вперед, одним из тех, которые помогут вам в написании красивого программного кода.

Благодарности

Мне хочется поблагодарить всех, кто помог мне улучшить содержимое этой главы, предоставив свои отзывы: Бо Адлера (*Bo Adler*), Джастина Бэйли (*Justin Bailey*), Мэттью Брекнелла (*Matthew Brecknell*), Пола Брауна (*Paul Brown*), Конала Эллиота (*Conal Elliot*), Тони Финча (*Tony Finch*), Катлина Фишера (*Kathleen Fisher*), Грэга Фицджеральда (*Greg Fitzgerald*), Бенжамина Франксена (*Benjamin Franksen*), Джереми Гиббонаса (*Jeremy Gibbons*), Тима Харриса (*Tim*

¹ Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, «Composable memory transactions», ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05), June 2005; Tim Harris & Simon Peyton Jones, «Transactional memory with data invariants», First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANS-ACT '06), Ottawa, June 2006, ACM; Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones & Satnam Singh, «Lock-free data structures using STMs in Haskell», Eighth International Symposium on Functional and Logic Programming (FLOPS '06), Apr. 2006.

² James Larus & Ravi Rajwar, «Transactional Memory», Morgan & Claypool, 2006.

Harris), Роберта Хелгессона (Robert Helgesson), Дина Херрингтона (Dean Herington), Дэвида Хауса (David House), Брайана Халлея (Brian Hulley), Дэйла Джордана (Dale Jordan), Марникса Клоостера (Marnick Klooster), Криса Куkleвича (Chris Kuklewicz), Эвана Мартина (Evan Martin), Грэга Мередита (Greg Meredith), Нейла Митчелла (Neil Mitchell), Джуну Мукаи (Jun Mukai), Михала Палка (Michal Palka), Себастьяна Сильвана (Sebastian Sylvan), Джоан Тибелл (Johan Tibell), Артура Ван Лейвена (Arthur van Leeuwen), Вима Вандербаухеде (Wim Vanderbauwheide), Дэвида Уакелинга (David Wakeling), Дэна Вана (Dan Wang), Петера Василко (Peter Wasilko), Эрика Уиллингерса (Eric Willigers), Гаала Яхаса (Gaal Yahas) и Брайана Зиммера (Brian Zimmer). Моя особая благодарность Кирстен Шевалье (Kirsten Chevalier), Энди Ораму (Andy Oram) и Грэгу Вилсону (Greg Wilson) за их очень подробные рецензии.

25 Синтаксическая абстракция: расширитель `syntax-case`

R. Кент Дибиг (R. Kent Dybvig)

При написании компьютерных программ постоянно возникают одни и те же типовые задачи. К примеру, в программах часто приходится осуществлять циклический перебор элементов массива, увеличивать или уменьшать значения переменных и многократно выполнять условные ветвления на основе числовых или символьных значений. Как правило, разработчики языков программирования признают это, включая в язык специально предназначенные синтаксические конструкции, которые обрабатывают наиболее типичные задачи. К примеру, язык Си предоставляет сразу несколько конструкций для построения цикла, для условного ветвления и для увеличения значений и иных обновлений значения переменной¹.

Некоторый типовые задачи менее распространены, но могут довольно часто встречаться в определенном классе программ или в какой-то отдельной программе. О существовании подобных задач разработчики языка могут даже и не догадываться, но в любом случае они, как правило, не склонны включать в ядро языка синтаксические конструкции для работы с ними.

И все же признавая, что такие типовые задачи имеют место, а специализированные синтаксические конструкции могут одновременно и упростить программы, и облегчить их чтение, создатели языка иногда включают в него механизмы, предназначенные для синтаксических абстракций, к которым относятся макросы препроцессора языка Си или макросы Общего Lisp (Common Lisp)². Когда такие возможности отсутствуют или не отвечают потребностям специального применения, могут быть пущены в ход внешние инструментальные средства, к примеру, макрорасширитель *m4*³.

¹ Brian W. Kernighan and Dennis M. Ritchie, «The C Programming Language», Second Edition, Prentice Hall, 1988.

² Guy L. Steele Jr., «Common Lisp: The Language», Second Edition, Digital Press, 1990.

³ Brian W. Kernighan and Dennis M. Ritchie, «The M4 Macro Processor», 1979.

Средства синтаксической абстракции различаются некоторыми специфическими подходами к решению своей задачи. Макрос препроцессора языка Си по своей сути построен на обработке лексем и позволяет заменять макровызовы последовательностью этих лексем; текст, переданный макровызову, заменяется на формальные параметры макроса, если таковые существуют. Макрос языка Lisp построен на выражениях и позволяет заменять отдельное выражение другим выражением, вычисляемым самим Lisp'ом и основанным на подвыражениях макровызова, если таковые существуют.

В обоих случаях идентификаторы, фигурирующие внутри подвыражения макровызова, имеют область действия там, где они появляются на выходе, а не там, где они появляются на входе, что может привести к непреднамеренному захвату ссылки на переменную за счет ее связывания.

Рассмотрим, к примеру, простое преобразование имеющегося в языке Scheme выражения `or1` в `let` и `if`, показанное в следующем примере:

```
(or e1 e2) → (let ([t e1]) (if t t e2))
```

ПРИМЕЧАНИЕ

Читатели, не знакомые с языком Scheme, могут обратиться к нескольким первым главам книги «The Scheme Programming Language», Third Edition (R. Kent Dybvig, MIT Press), которая выложена по адресу: <http://www.scheme.com/tspl3>.

Выражение `or` должно вернуть значение своего первого подвыражения, если оно вычисляется как истинное (любое неложное) значение. Выражение `let` используется, чтобы дать имя этому значению и не вычислять его дважды.

Предыдущее преобразование в большинстве случаев со своей задачей справляется, но оно не работает, если идентификатор `t` появляется в выражении `e2` свободно (то есть за пределами любого связывания `t` в `e2`), что и происходит в следующем выражении:

```
(let ([t #t]) (or #f t)),
```

Оно будет вычислено в истинное значение `#t`. Если взять ранее определенное простое преобразование `or`, то выражение будет развернуто в следующее выражение:

```
(let ([t #t])
  (let ([t #f])
    (if t t t))),
```

которое будет вычислено в ложное значение `#f`.

Обнаружив эту проблему, ее нетрудно решить, воспользовавшись генерированным идентификатором для введенного связывания:

```
(or e1 e2) → (let ([g e1]) (if g g e2))
```

где `g` — это генерированный (новый) идентификатор.

¹ Richard Kelsey, William Clinger, and Jonathan Rees, «Revised report on the algorithmic language Scheme», Higher-Order and Symbolic Computation, vol. 11, № 1, pp. 7–105, 1998. Также появляясь в ACM SIGPLAN Notices, vol. 33, № 9, Sep. 1998.

Как отмечали в своей оригинальной статье, посвященной гигиеническому макрорасширению¹, Колбеккер (Kohlbecker), Фридман (Friedman), Фелайсен (Felleisen) и Дюба (Duba), подобные проблемы отличаются особым коварством, поскольку преобразование может безупречно работать в большом объеме программного кода, а позже взять и отказать, при этом сильно затруднив отладку.

Если непреднамеренные захваты, вызванные *связываниями* введенных идентификаторов, всегда могут быть обойдены за счет использования сгенерированных идентификаторов, то для *ссылок* на введенные идентификаторы, которые могут быть захвачены за счет связываний контекста макровызова, такого же простого решения не существует. В следующем примере *if* имеет лексическую связь в контексте выражения ог:

```
(let ([if (lambda (x y z) "oops")]) (or #f #f)).
```

При вторичном преобразовании ог это выражение расширяется в следующее:

```
(let ([if (lambda (x y z) "oops")])
  (let ([g #f])
    (if g g #f))),
```

где *g* является новым идентификатором. Значение выражения должно быть *#f*, но на самом деле оно будет равно «oops», поскольку локально связанная процедура *if* используется вместо оригинального синтаксиса условия *if*.

Установка в языке ограничений за счет резервирования таких ключевых слов, как *let* и *if*, решит эту проблему в отношении этих слов, но не решит проблему в целом. К примеру, такая же ситуация может возникнуть с введенной ссылкой на определенную пользователем переменную *add1* в следующем преобразовании *increment*:

```
(increment x) → (set! x(add1 x)).
```

Колбеккер и компания для решения обеих разновидностей проблем захвата придумали концепцию *гигиенического* макрорасширения, позаимствовав термин «гигиена» у Барендргта (Barendregt)². Введенные Барендргтом гигиенические условия для λ -вычислений придерживаются предположения, что свободные переменные одного выражения, заменяемые на другие переменные, не захватываются связыванием другим выражением, пока в таком захвате не возникнет явная необходимость. Колбеккер и компания адаптировали все это в следующие *гигиенические условия для макрорасширения*:

Сгенерированные идентификаторы, ставшие экземплярами связывания в полностью расширенной программе, должны связывать только те переменные, которые были сгенерированы на том же этапе транскрибирования.

На практике это требование заставляет расширитель осуществлять переименование идентификаторов по мере необходимости, чтобы избежать непреднамеренных захватов. Например, если взять оригинальное преобразование ог:

```
(or e1 e2) → (let ([t e1]) (if t t e2))
```

¹ Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen u Bruce Duba, «Hygienic macro expansion», Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, pp. 151–161, 1986.

² H. P. Barendregt, «Introduction to the lambda calculus», Nieuw Archief voor Wisenkunde, vol. 4, № 2, pp. 337–372, 1984.

то выражение:

```
(let ([t #t]) (or #f t))
```

расширяется в эквивалент:

```
(let ([t0 #t])
  (let ([t1 #f])
    (if t1 t1 t0)))
```

который вполне корректно вычисляется в #t. Таким же образом выражение:

```
(let ([if (lambda (x y z) "oops")]) (or #f #f))
```

расширяется в эквивалент:

```
(let ([if0 (lambda (x y z) "oops")])
  (let ([t #f])
    (if t t #f)))
```

который корректно вычисляется в #f.

По сути, гигиеническое макрорасширение осуществляет лексический обзор исходного кода, тогда как негигиеническое расширение осуществляет лексический обзор кода после расширения.

Гигиеническое расширение может сохранить лексическую область видимости только в тех пределах, в которых эта область сохраняется теми преобразованиями, которые ему предписано выполнить. Но преобразование может создать код, который вторгается в лексическую область видимости явным образом. В качестве иллюстрации можно привести следующее (некорректное) преобразование let:

```
(let ((x e)) тело) → (letrec ((x e)) тело)
```

Выражение e должно появиться вне области видимости связывания переменной x, но на выходе, благодаря семантике letrec, оно появляется внутри этой области.

Алгоритм гигиенического макрорасширения (KFFD), определенный Колбеккером и компанией, имеет остроумное и изящное описание. Он работает за счет добавления меток времени к каждой переменной, представленной макросом, и их последующего использования для того, чтобы различать переменные с одинаковыми именами по мере того, как он осуществляет переименование лексически связанных переменных. Но у этого алгоритма есть ряд недостатков, которые на практике препятствуют его непосредственному применению. Самый серьезный из них — это отсутствие поддержки в макросе локальных связей и существенные издержки, возникающие из-за полного переписывания каждого выражения при установке меток времени и переименовании.

Эти недостатки можно отнести на счет системы syntax-rules, разработанной Клингером (Clinger), Дибвигом (Dybvig), Хибом (Hieb) и Рисом (Rees) для исправленного описания языка Scheme¹. Простота syntax-rules, системы, основанной на сопоставлении шаблонов (pattern-based), позволяет без особого труда

¹ William Clinger and Jonathan Rees, «Revised report on the algorithmic language Scheme», LISP Pointers, vol. 4, № 3, pp. 1–55, Jul.–Sep. 1991.

добиться ее эффективной реализации¹. К сожалению, эта же простота ограничивает универсальность механизма, поэтому многие полезные синтаксические абстракции создать очень трудно или вовсе невозможно.

Система syntax-case² была разработана для устранения недостатков, присущих исходному алгоритму без введения ограничений на syntax-rules. Система поддерживает локальные связывания в макросах и работает с постоянными ограничениями, все же позволяющими макросам использовать всю впечатляющую мощь языка Scheme. Она совместима снизу вверх с системой syntax-rules, которая может быть выражена в виде простого макроса на языке syntax-case, и она позволяет использовать тот же язык шаблонов даже для «низкоуровневых» макросов, для которых syntax-rules применяться не могут. Она также предоставляет механизм, который разрешает использование *умышленных* захватов — то есть позволяет «отклонить» или «нарушить» гигиену простым и избирательным способом. Кроме того, она справляется с рядом практических аспектов расширения, без которых не обойтись в реальной разработке, к которым относятся внутренние определения и отслеживание исходной информации через макрорасширение.

За все это приходится платить сложностью алгоритма расширения и размером кода, необходимого для его реализации. Поэтому изучение полной реализации во всем ее великолепии выходит за рамки этой главы. Вместо этого мы исследуем упрощенную версию расширителя, которая послужит иллюстрацией для основного алгоритма и наиболее важных аспектов его реализации.

Краткое введение в syntax-case

Начнем с нескольких небольших примеров syntax-case, позаимствованных из «Chez Scheme Version 7 User's Guide» (R. Kent Dybvig, Cadence Research Systems, 2005). Другие примеры и более подробное описание syntax-case приведены в том же источнике, и в книге «The Scheme Programming Language», Third Edition.

Следующее определение ог иллюстрирует форму макроопределения syntax-case:

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2)
       (syntax (let ([t e1]) (if t t e2))))]))
```

Выражение define-syntax создает связывание ключевого слова, ассоциируя в данном случае ключевое слово ог с процедурой преобразования, или преобра-

¹ William Clinger and Jonathan Rees, «Macros that work», Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, pp. 155–162, Jan. 1991.

² R. Kent Dybvig, Robert Hieb and Carl Bruggeman, «Syntactic abstraction in Scheme», Lisp and Symbolic Computation, vol. 5, № 4, pp. 295–326, 1993.

зователем. Преобразователь действует за счет вычисления, производимого во время расширения lambda-выражения в правой части выражения define-syntax. Выражение syntax-case используется для анализа входных данных, а выражение syntax используется для создания выходных данных путем простого соответствия шаблону. Шаблон (_e1 e2) определяет выражение входных данных, где знаком подчеркивания (_) помечено место появления ключевого слова or, а переменные шаблона e1 и e2 связаны с первым и вторым подчиненными выражениями. Шаблон let ([t e1]) (if t e2)) определяет выражение выходных данных, в котором e1 и e2 вставляются из входных данных.

Выражение (syntax шаблон) может быть сокращено до #'шаблон, поэтому предыдущее определение может быть переписано следующим образом:

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2)
       #'(let ([t e1]) (if t e2))))))
```

Макрос также может быть ограничен в пределах одного выражения с помощью letrec-syntax.

```
(letrec-syntax ([or (lambda (x)
                      (syntax-case x ()
                        [(_ e1 e2)
                         #'(let ([t e1]) (if t e2))))])
  (or a b))
```

Макрос может быть рекурсивным (то есть расширяться в свои собственные появления), как показано в следующей версии or, оперирующей произвольным числом подчиненных выражений. Для управления двумя основными и рекурсивным экземпляром требуются несколько операторов syntax-case:

```
(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ #'f)
       [(_ e) #'e]
       [(_ e1 e2 e3 ...)
        #'(let ([t e1]) (if t t (or e2 e3 ...))))]))
```

На языке шаблона syntax-case выражение входных или выходных данных с последующим многоточием приводит к проверке на соответствие или к созданию от нуля и более выражений.

В этом примере для определений or гигиена гарантирована, так что введенное связывание для t и введенные ссылки для let, if и даже для or имеют соответствующую область действия. Если мы хотим отклонить или нарушить гигиену, то можем добиться этого с помощью процедуры datum->syntax, которая создает объект синтаксиса из произвольного s-выражения. Идентификаторы, принадлежащие s-выражению, рассматриваются, как будто они появились в оригинальном исходном коде, там, где появился первый аргумент — идентификатор шаблона.

Мы можем воспользоваться этим обстоятельством для создания простой синтаксической структуры `method`, которая неявным образом связывает имя `this` с первым (дополнительным) аргументом:

```
(define-syntax method
  (lambda (x)
    (syntax-case x ()
      [(k (x ...) e1 e2 ...)
       (with-syntax ([this (datum->syntax #'k 'this)])
         #'(lambda (this x ...) e1 e2 ...))))))
```

За счет использования ключевого слова `k`, извлекаемого из входных данных в качестве переменной шаблона, переменная `this` рассматривается так, будто она присутствует в выражении `method`, так что:

```
(method (a) (f this a))
```

рассматривается как эквивалент следующего выражения:

```
(lambda (this a) (f this a))
```

без переименования, чтобы уберечь введенное связывание от захвата ссылки исходного кода.

Выражение `with-syntax`, примененное в определении `method`, создает локальное связывание переменной шаблона. Оно представляет собой простой макрос, написанный на языке `syntax-case`:

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      [(_ ((p e0 ...) e1 e2 ...))
       #'(syntax-case (list e0 ...) ()
                     [(_ ...) (begin e1 e2 ...)]))]))
```

Процедура `datum->syntax` может быть использована для произвольных выражений, что можно проиллюстрировать следующим определением `include`:

```
(define-syntax include
  (lambda (x)
    (define read-file
      (lambda (fn k)
        (let ([p (open-input-file fn)])
          (let f ([x (read p)])
            (if (eof-object? x)
                (begin (close-input-port p) '())
                (cons (datum->syntax k x) (f (read p))))))))
    (syntax-case x ()
      [(k filename)
       (let ([fn (syntax->datum #'filename)])
         (with-syntax ([(_ ...) (read-file fn #'k)])
           #'(begin _ ...))))]))
```

Выражение (`include «имя_файла»`) приводит к рассмотрению выражений внутри названного файла, как будто бы они присутствуют в исходном коде на месте выражения `include`. В дополнение к использованию `datum->syntax` выражение `include` также использует его обратный оператор `syntax->datum` для преобразования подвыражения с именем файла в строку, которую оно может передать в `open-input-file`.

Алгоритм расширения

Алгоритм расширения syntax-case по сути является «ленивым» вариантом алгоритма KFFD, который оперирует абстрактным представлением входного выражения вместо традиционного представления s-выражения. Абстрактное представление инкапсулирует как представление выражения входных данных, так и *оболочку*, позволяющую алгоритму определить область действия всех идентификаторов внутри выражения. Оболочка состоит из *маркеров* и *подстановок*.

Маркеры аналогичны меткам времени KFFD и добавляются к частям выходных данных, вставляемых макроМандой.

Подстановки отображают идентификаторы связывания с помощью среды, создаваемой во время компиляции. Подстановки создаются при каждом вычислении связанного выражения, такого как `lambda`, и они добавляются к оболочкам объектов синтаксиса, представляющих выражения внутри области действия связывания, существующей в связанных выражениях. Подстановки применяются к идентификатору, только если идентификатор имеет такое же имя и помечен как идентификатор для подстановки.

Расширение действует рекурсивно, сверху вниз. Как только расширитель сталкивается с макроВызовом, он вызывает для выражения соответствующий преобразователь, сначала отмечая его новым маркером, а затем отмечая его снова тем же самым маркером. Одинаковые маркеры отменяются, поэтому остаются помеченными только вставленные части продукции макроса — то есть те части, которые не были просто скопированы из входа в выход.

Когда встречается основное выражение, то создается основное выражение на выходном языке расширителя (в нашем случае это обычное представление s-выражения), при необходимости со всеми рекурсивно расширенными подвыражениями. Ссылки на переменные заменяются именами, генерированными механизмом подстановки.

Представления

Наиболее важным аспектом механизма syntax-case является его абстрактное представление кода исходной программы как *объектов синтаксиса*. Ранее мы уже выяснили, что объекты синтаксиса инкапсулируют не только представление исходного программного кода, но также являются оболочкой, которая предоставляет достаточную информацию о содержащихся в коде идентификаторах, с помощью которых осуществляется гигиена:

```
(define-record syntax-object (expr wgar))
```

Выражение `define-record` создает новый тип значения с указанным именем (в данном случае `syntax-object`) и полями (в данном случае `expr` и `wgar`), наряду с набором процедур для работы с ними. В данном случае речь идет о следующих процедурах:

`make-syntax-object`

Возвращает новый объект синтаксиса, устанавливая полям `expr` и `wgar` значения своих аргументов

`syntax-object?`

Возвращает `true` только в том случае, если ее аргументы являются объектами синтаксиса

`syntax-object-expr`

Возвращает значение поля `expr` объекта синтаксиса

`syntax-object-wrap`

Возвращает значение поля `wrap` объекта синтаксиса

Полная реализация `syntax-case` могла бы также включать внутри каждого объекта синтаксиса исходную информацию, отслеживаемую в течение процесса расширения.

Мы уже выяснили, что каждая оболочка состоит из списка маркеров и подстановок. Маркеры различаются по своей идентичности объекту и не требуют никаких полей:

```
(define-record mark ())
```

Подстановки отображают символическое имя и перечисление от маркеров до метки:

```
(define-record subst (sym mark* label))
```

Метки, как и маркеры, различаются по своей идентичности и не требуют полей:

```
(define-record label ())
```

Среди процесса расширения, поддерживаемая расширителем, проецирует метки на *связывания*. Среда структурируется как обычный *ассоциативный список* — то есть список пар, где каждый первый элемент *cadr* является меткой, а остальная часть *cdr* является связыванием. Связывания состоят из типа (представленного обозначением) и значения (соответственно, *type* и *value*):

```
(define-record binding (type value))
```

Тип определяет природу связывания: *macro* — для связывания ключевых слов и *lexical* — для связываний переменной *lexical*. Значение представляет собой любую дополнительную информацию, необходимую для определения связывания, такую как процедуру преобразования, когда связывание имеет отношение к ключевым словам.

Создание выходных данных расширителя

Выход расширителя — это простое *s*-выражение на основном языке, и поэтому оно главным образом построено с использованием синтаксиса квазицитирования языка Scheme для создания списковой структуры. Например, *lambda*-выражение может быть создано с формальным параметром *var* и телом *body*:

```
(lambda (.var) .body)
```

Но расширителю нужно создавать новые имена и делать это посредством помощника *gen-var*, который использует базисные элементы Scheme для преобразования строк в обозначения и наоборот, наряду с локальным счетчиком последовательности:

```
(define gen-var
  (let ([n 0])
    (lambda (id)
      (set! n (+ n 1))
      (let ([name (syntax-object-expr id)])
        (string->symbol (format "-s.-s" name n))))))
```

Вскрытие объектов синтаксиса

При каждой встрече на входе выражения `quote`, расширитель должен вернуть представление неизменного содержимого, появляющегося внутри выражения `quote`. Для этого он должен вскрыть любые вложенные объекты синтаксиса и оболочки, использовав процедуру `strip`, которая проходит по объекту синтаксиса, составляет список структуры его входных данных и восстанавливает *s*-выражения, представленные на входе:

```
(define strip
  (lambda (x)
    (cond
      [(syntax-object? x)
       (if (top-marked? (syntax-object-wrap x))
           (syntax-object-expr x)
           (strip (syntax-object-expr x)))]
      [(pair? x)
       (let ([a (strip (car x))] [d (strip (cdr x))])
         (if (and (eq? a (car x)) (eq? d (cdr x)))
             x
             (cons a d)))]
      [else x])))
```

Прохождение по любой из ветвей входного выражения завершается, когда будет найдено что-либо отличное от объекта синтаксиса или пары — то есть когда найдено обозначение или непосредственное значение. Оно также завершается, когда обнаруживается, что объект синтаксиса «помечен как вершина» — то есть когда его оболочка содержит уникальный маркер *top mark*:

```
(define top-mark (make-mark))

(define top-marked?
  (lambda (wrap)
    (and (not (null? wrap))
         (or (eq? (car wrap) top-mark)
             (top-marked? (cdr wrap))))))
```

Когда расширитель создает объект синтаксиса, представляющий первоначальные входные данные, он использует оболочку, которая на дне содержит маркер вершины — *top mark*, именно для того чтобы позволить коду вскрытия определить, когда он достиг дна объекта синтаксиса и уже не должен идти по этому объекту дальше. Это свойство удерживает расширитель от ненужного прохождения неизменяемых составляющих, с тем чтобы надежно уберечь общие и циклические структуры, и не позволяет быть сбитым с толку присутствием цитируемым объектов синтаксиса во входных данных.

Синтаксические ошибки

Расширитель сообщает об ошибках синтаксиса через объект syntax-error, который определяется следующим образом:

```
(define syntax-error
  (lambda (object message)
    (error #f "-a ~s" message (strip object))))
```

Если реализация прилагает к объектам синтаксиса исходную информацию, то эта информация может быть использована для построения сообщения об ошибке, которое содержит в себе исходную строку и позицию символа.

Структурные предикаты

Неатомарная структура объекта синтаксиса всегда определяется образцами выражения syntax-case. Предикат identifier? определяет, представляет ли объект синтаксиса идентификатор:

```
(define identifier?
  (lambda (x)
    (and (syntax-object? x)
         (symbol? (syntax-object-expr x)))))
```

Точно так же предикат self-evaluating? используется после вскрытия объекта синтаксиса для определения, представляет ли он неизменяемую составляющую:

```
(define self-evaluating?
  (lambda (x)
    (or (boolean? x) (number? x) (string? x) (char? x))))
```

Создание оболочек

Маркеры и подстановки добавляются к объекту синтаксиса за счет расширения оболочки:

```
(define add-mark
  (lambda (mark x)
    (extend-wrap (list mark) x)))
(define add-subst
  (lambda (id label x)
    (extend-wrap
      (list (make-subst
        (syntax-object-expr id)
        (wrap-marks (syntax-object-wrap id))
        label))
      x)))
```

Если объект синтаксиса заключен в оболочку лишь частично, оболочка расширяется просто за счет создания объекта синтаксиса, инкапсулирующего частично помещенную в оболочку структуру. В противном случае объект синтаксиса реконструируется, и новая оболочка присоединяется к старой:

```
(define extend-wrap
  (lambda (wrap x)
```

```
(if (syntax-object? x)
  (make-syntax-object
    (syntax-object-expr x)
    (join-wraps wrap (syntax-object-wrap x)))
  (make-syntax-object x wrap)))
```

Соединение двух оболочек по сути не сложнее, чем дополнение списка маркеров. Единственная сложность заключается в том, что алгоритм расширения требует, чтобы два одинаковых маркера при встрече аннулировались.

```
(define join-wraps
  (lambda (wrap1 wrap2)
    (cond
      [(null? wrap1) wrap2]
      [(null? wrap2) wrap1]
      [else
        (let f ([w (car wrap1)] [w* (cdr wrap1)])
          (if (null? w*)
              (if (and (mark? w) (eq? (car wrap2) w))
                  (cdr wrap2)
                  (cons w wrap2))
                  (cons w (f (car w*) (cdr w*))))))))]))
```

Управление средами

Среды отображают метки на связывание и представлены в виде ассоциированных списков. Поэтому расширение среды затрагивает добавление отдельной пары, отображающей метку на связывание:

```
(define extend-env
  (lambda (label binding env)
    (cons (cons label binding) env)))
```

Анализ идентификатора

Определение связывания, ассоциированного с идентификатором, происходит в два этапа. Сначала определяется метка, ассоциированная с идентификатором в оболочке идентификатора, а потом ведется поиск метки в текущей среде:

```
(define id-binding
  (lambda (id r)
    (label-binding id (id-label id) r)))
```

Маркеры и подстановки, появляющиеся в оболочке идентификаторов, определяют ассоциированную метку, если таковая имеется. Подстановки отображают имена и списки от маркеров до меток. Любая подстановка, чье имя не является именем идентификатора, игнорируется, как и любая другая, чьи маркеры не совпадают. Имена являются символами, и поэтому они сравниваются с помощью оператора эквивалентности указателей — `eq?`.

Соответствующим считается такой набор маркеров, который наслался бы на оболочку перед подстановкой. Таким образом, набор маркеров, с которым сравниваются маркеры подстановок, изменяется по мере продвижения поиска по оболочке. Начальным набором маркеров является полный набор, фигурирующий

в оболочке. Как только маркер встречается в процессе поиска соответствующей подмены в оболочке, первый маркер в списке удаляется:

```
(define id-label
  (lambda (id)
    (let ([sym (syntax-object-expr id)]
          [wrap (syntax-object-wrap id)])
      (let search ([wrap wrap] [mark* (wrap-marks wrap)])
        (if (null? wrap)
            (syntax-error id "неопределенный идентификатор")
            (let ([w0 (car wrap)])
              (if (mark? w0)
                  (search (cdr wrap) (cdr mark*))
                  (if (and (eq? (subst-sym w0) sym)
                           (same-marks? (subst-mark* w0) mark*))
                      (subst-label w0)
                      (search (cdr wrap) mark*)))))))))
```

Если в оболочке не имеется ни одной соответствующей подстановки, идентификатор не определяется, и подается сигнал о синтаксической ошибке. Вместо этого можно было бы рассматривать все подобные ссылки на идентификаторы как ссылки на глобальные переменные.

Процедура `id-label` получает первоначальный список маркеров через `wrap-marks` и использует предикат `same-marks?` для сравнения списков маркеров:

```
(define wrap-marks
  (lambda (wrap)
    (if (null? wrap)
        '()
        (let ([w0 (car wrap)])
          (if (mark? w0)
              (cons w0 (wrap-marks (cdr wrap)))
              (wrap-marks (cdr wrap)))))))

(define same-marks?
  (lambda (m1* m2*)
    (if (null? m1*)
        (null? m2*)
        (and (not (null? m2*))
             (eq? (car m1*) (car m2*))
             (same-marks? (cdr m1*) (cdr m2*))))))
```

Как только метка будет найдена, `id-binding` используется для поиска ассоциированного связывания, если таковое имеется, используя процедуру `assq` для поиска списков ассоциаций. Если ассоциации найдены, то возвращается связывание в оставшейся части ассоциации — `cdr`:

```
(define label-binding
  (lambda (id label r)
    (let ([a (assq label r)])
      (if a
          (cdr a)
          (syntax-error id "лексически отклоненный")))))
```

Если связывание не найдено, идентификатор является «лексически отклоненным». Это случается, когда макрос ошибочно вставляет в свои выходные

данные ссылку на идентификатор, который не виден в контексте его выходных данных.

Расширитель

При наличии механизмов обработки оболочек и сред работы расширителя не представляет особой сложности. Расширитель выражения — `exp` обрабатывает макровызовы, ссылки на лексические переменные, приложения, основные выражения и неизменяемые компоненты. Макровызовы поступают в двух формах: в виде одиночных ссылок на ключевые слова макроса и в виде структурированных выражений, содержащих в начальной позиции ключевое слово макроса.

Процедура `exp` воспринимает три аргумента: синтаксический объект `x`, среду времени выполнения `r` и метасреду `mr`. Среда времени выполнения используется для обработки обычных выражений, чей код появится в выходных данных расширителя, а метасреда используется для обработки преобразуемых выражений (к примеру, тех, что справа от `letrec-syntax-связываний`), которые вычисляются и используются во время расширения. Разница между средой времени выполнения и метасредой состоит в том, что метасреда не содержит связываний лексических переменных, поскольку такие связывания не доступны при использовании преобразователя и проведении необходимых ему вычислений:

```
(define exp
  (lambda (x r mr)
    (syntax-case x ()
      [id
       (identifier? #'id)
       (let ([b (id-binding #'id r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical) (binding-value b)]
           [else (syntax-error x "неверный синтаксис ")])))
      [(e0 el ...)
       (identifier? #'e0)
       (let ([b (id-binding #'e0 r)])
         (case (binding-type b)
           [(macro) (exp (exp-macro (binding-value b) x) r mr)]
           [(lexical)
            `(.(binding-value b) ,(exp-exprs #'(el ...) r mr))]
           [(core) (exp-core (binding-value b) x r mr)]
           [else (syntax-error x "неверный синтаксис ")]))]
      [(e0 el ...)
       `(.(exp #'e0 r mr) ,(exp-exprs #'(el ...) r mr))]
      [_
       (let ([d (strip x)])
         (if (self-evaluating? d)
             d
             (syntax-error x "неверный синтаксис "))))]))
```

Макровызовы обрабатываются процедурой `exp-macro` (краткое описание которой приводится), а затем повторно расширяются. Лексические переменные переписываются в связанное значение, которое всегда является генерированным

именем переменной. Приложения переписываются в списки, как и в обычном синтаксисе s-выражений для языков Lisp и Scheme, с рекурсивным расширением подчиненных выражений. Основные выражения обрабатываются процедурой `exp-core` (краткое описание которой приводится); любая рекурсия, обращенная назад к расширителю выражения, выполняется в прямом виде основным преобразователем. Неизменяемая часть переписывается в постоянное значение, извлеченное из своей синтаксической оболочки.

Расширитель использует `syntax-case` и `syntax` (в сокращенной форме – #'шаблон) для анализа и ссылок на входные данные или части таковых. Поскольку расширитель также заполнен реализациями `syntax-case`, это может показаться парадоксом. В действительности обработка идет за счет начальной загрузки одной версии расширителя с использованием предыдущей версии. Если не использовать `syntax-case` и `syntax`, то написать расширитель было бы намного труднее.

Процедура `exp-macro` применяет процедуру преобразования (совокупность значений макроссвязывания) ко всему макровыражению, которое может быть либо одиночным ключевым словом макроса, либо структурированным выражением, начинающимся с ключевого слова макроса. Сперва процедура `exp-macro` добавляет новый маркер к оболочке входного выражения, затем применяет тот же маркер к оболочке выходного выражения. Первый маркер служит в качестве «антимаркера», который отменяет второй маркер, создавая тем самым результирующий эффект присоединения маркера только к тем частям выходных данных, которые были представлены преобразователем, однозначно идентифицируя таким образом части кода, представленные на этом этапе транскрибирования:

```
(define exp-macro
  (lambda (p x)
    (let ([m (make-mark)])
      (add-mark m (p (add-mark m x))))))
```

Процедура `exp-core` просто применяет заданный основной преобразователь (совокупность значений основного связывания) к входному выражению:

```
(define exp-core
  (lambda (p x r mr)
    (p x r mr)))
```

Процедура `exp-exprs` используется для обработки прикладных подчиненных выражений простым отображением на них расширителя:

```
(define exp-exprs
  (lambda (x* r mr)
    (map (lambda (x) (exp x r mr)) x*)))
```

Основные преобразователи

Здесь описываются преобразователи для некоторых типичных основных выражений (`quote`, `if`, `lambda`, `let` и `letrec-syntax`). Добавление преобразователей для других основных выражений, таких как `letrec` или `let-syntax`, особых трудностей не вызывает.

Процедура `exp-quote` вырабатывает *s*-выражения, представляющие выражения `quote`, со значениями данных, освобожденных от их синтаксической оболочки:

```
(define exp-quote
  (lambda (x r mr)
    (syntax-case x ()
      [(_ d) `(quote ,(strip #'d))))))
```

Процедура `exp-if` вырабатывает *s*-выражения, представляющие выражение `if`, с рекурсивно расширяемыми подвыражениями:

```
(define exp-if
  (lambda (x r mr)
    (syntax-case x ()
      [(_ e1 e2 e3)
       `(if ,(exp #'e1 r mr)
            ,(exp #'e2 r mr)
            ,(exp #'e3 r mr))))))
```

Процедура `exp-lambda` обрабатывает `lambda`-выражения, у которых имеется только один формальный параметр и только одно тело выражения. Ее расширение для обработки нескольких параметров особого труда не представляет. Куда сложнее обработка произвольных `lambda`-тел, включая внутренние определения, но поддержка внутренних определений выходит за рамки материала данной главы.

При выработке представления `lambda`-выражения в виде *s*-выражения для формального параметра создается генерированное имя переменной. Подстановка отображает идентификатор на новую метку, добавленную к оболочке тела, и среда расширяется ассоциацией от метки к связыванию `lexical`, чье значение является переменной, генерированной во время рекурсивной обработки тела:

```
(define exp-lambda
  (lambda (x r mr)
    (syntax-case x ()
      [(_ (var) body)
       `(let ([label (make-label)] [new-var (gen-var #'var)])
          `(lambda ,(new-var)
             ,(exp (add-subst #'var label #'body)
                   (extend-env label
                               (make-binding 'lexical new-var)
                               r)
                   mr))))]))
```

Метасреда не подвергается расширению, поскольку она не должна включать связывания лексических переменных.

Процедура `exp-let`, занимающаяся преобразованием выражений `let`, имеющих одиночное связывание, похожа на несколько более запутанный преобразователь для `lambda`:

```
(define exp-let
  (lambda (x r mr)
    (syntax-case x ()
      [(_ ([var expr]) body)
```

```
(let ([label (make-label)] [new-var (gen-var #'var)])
  (let ([.new-var .(exp #'expr r mr)])
    ,(exp (add-subst #'var label #'body)
          (extend-env label
                      (make-binding 'lexical new-var)
                      r)
          mr))))]))
```

Тело находится в области действия связывания, созданного let, поэтому оно расширяется с расширением оболочки и среды. Выражение, расположенное справа, — expr не находится в области действия, поэтому оно расширяется с использованием первоначальной оболочки и среды.

Процедура exp-letrec-syntax обрабатывает выражения letrec-syntax, имеющие одно связывание. Как и в случае с lambda и let, подстановка отображает связанный идентификатор — в данном случае это ключевое слово, а не переменная — на новую метку, которая добавляется к оболочке тела, а ассоциация от метки к связыванию добавляется к среде при рекурсивной обработке тела. Связывание относится к макроссвязыванию, а не к lexical-связыванию, и значение связывания является результатом рекурсивного расширения и вычисления правой части выражения letrec-syntax.

В отличие от let, правая часть выражения также заключена в оболочку подстановкой от ключевого слова к метке и расширяется с расширяемой средой; это позволяет макросу быть рекурсивным. Если выражение было бы не letrec-syntax, а let-syntax, то этого бы не получилось. Выходные данные, вырабатываемые за счет расширения letrec-syntax, состоят только из выходных данных рекурсивного вызова расширителя тела выражения:

```
(define exp-letrec-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_) ((kwd expr)) body]
      [(let ([label (make-label)])]
       (let ([b (make-binding 'macro
                             (eval (exp (add-subst #'kwd label #'expr)
                                         mr mr))))])
         (exp (add-subst #'kwd label #'body)
              (extend-env label b r)
              (extend-env label b mr))))]))
```

В данном случае расширению подвергаются и среда времени выполнения, и метасреда, поскольку преобразователи доступны и во время выполнения, и в коде преобразователя.

Анализ и создание объектов синтаксиса

Макросы пишутся в стиле сопоставления с образцом, используя для сопоставления syntax-case и производя разбор входных данных и используя syntax для воспроизведения выходных данных. Реализация сопоставления с образцом и воспроизведения выходит за рамки тематики данной главы, но следующие низкоуровневые операции могут быть использованы в качестве основы реализации. Выражение syntax-case может быть выстроено в виде следующего набора

из трех операторов, которые обрабатывают объекты синтаксиса как абстрактные s-выражения:

```
(define syntax-pair?
  (lambda (x)
    (pair? (syntax-object-expr x)))))

(define syntax-car
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)
      (car (syntax-object-expr x))))))

(define syntax-cdr
  (lambda (x)
    (extend-wrap
      (syntax-object-wrap x)
      (cdr (syntax-object-expr x))))))
```

В определениях syntax-car и syntax-cdr применяется вспомогательная функция extend-wrap, определенная в предыдущем разделе «Создание оболочек» для создания оболочки для пары из первого элемента — car и остальной части — cdr.

Аналогичным образом, syntax может быть построена из следующих, более простых версий syntax, которые обрабатывают неизменяемые составляющие входных данных, но не образцы переменных или многоточий:

```
(define exp-syntax
  (lambda (x r mr)
    (syntax-case x ()
      [(_ t) `(quote #',t)])))
```

По существу упрощенная версия syntax практически похожа на quote, за исключением того, что syntax не вскрывает инкапсулированное значение, вернее, оставляет оболочки синтаксиса нетронутыми.

Сравнение идентификаторов

Идентификаторы сравниваются по своему предназначению. Они могут сравниваться как символы с помощью оператора эквивалентности указателей — eq? по символическим именам идентификаторов. Они также могут пройти сравнение в соответствии с их использованием по предназначению как свободные или связанные идентификаторы в выходных данных макроса.

Два идентификатора считаются эквивалентными при использовании free-identifier=? , если они будут отнесены к одному и тому же связыванию, если вставлены в выходные данные макроса за пределами любого связывания, внешнего макросом. Эквивалентность проверяется путем сравнения меток, к которым отнесены идентификаторы, как описано в разделе «Анализ идентификатора»:

```
(define free-identifier?
  (lambda (x y)
    (eq? (id-label x) (id-label y))))
```

Предикат `free-identifier=?` часто используется для проверки на вспомогательные ключевые слова, такие как `else` в `cond` или `case`.

Два идентификатора считаются эквивалентными при использовании `bound-identifier=?`, если ссылка на один из них будет захвачена включенным связыванием для другого. Это достигается за счет сравнения имен и маркеров двух идентификаторов:

```
(define bound-identifier=?  
  (lambda (x y)  
    (and (eq? (syntax-object-expr x) (syntax-object-expr y))  
         (same-marks?  
          (wrap-marks (syntax-object-wrap x))  
          (wrap-marks (syntax-object-wrap y)))))))
```

Предикат `bound-identifier=?` часто используется для проверки на ошибки продублированных идентификаторов в выражении связывания, таких как `lambda` или `let`.

Преобразование

Преобразование *s*-выражения в объект синтаксиса выполняется процедурой `datum->syntax`, которой нужно, чтобы оболочка была передана от идентификатора шаблона *s*-выражению:

```
(define datum->syntax  
  (lambda (template-id x)  
    (make-syntax-object x (syntax-object-wrap template-id))))
```

Обратное преобразование вызывает удаление оболочки объекта синтаксиса, поэтому `syntax->datum` – это всего лишь `strip`:

```
(define syntax->datum strip)
```

Начало расширения

Теперь у нас есть все составляющие для расширения выражения Scheme, содержащего макросы внутри выражений, составленных на основном языке. Основной расширитель просто дает начальную оболочку и среду, включающую имена и связывания для основных выражений и примитивов:

```
(define expand  
  (lambda (x)  
    (let-values ([(wrap env) (initial-wrap-and-env)])  
      (exp (make-syntax-object x wrap) env env))))
```

Начальная оболочка состоит из набора подстановок, отображающих каждый предопределенный идентификатор на новую метку, и начальной среды, ассоциирующей каждую из этих меток с соответствующим связыванием:

```
(define initial-wrap-and-env  
  (lambda ()  
    (define id-binding*  
      `((quote . ,(make-binding 'core exp-quote))  
        (if . ,(make-binding 'core exp-if))  
        (lambda . ,(make-binding 'core exp-lambda))))
```

```

(let . .(make-binding 'core exp-let))
(letrec-syntax . .(make-binding 'core exp-letrec-syntax))
(identifier? . .(make-binding 'lexical 'identifier?))
(free-identifier=? . .(make-binding 'lexical 'free-identifier=?))
(bound-identifier=? . .(make-binding 'lexical 'bound-identifier=?))
(datum->syntax . .(make-binding 'lexical 'datum->syntax))
(syntax->datum . .(make-binding 'lexical 'syntax->datum))
(syntax-error . .(make-binding 'lexical 'syntax-error))
(syntax-pair? . .(make-binding 'lexical 'syntax-pair?))
(syntax-car . .(make-binding 'lexical 'syntax-car))
(syntax-cdr . .(make-binding 'lexical 'syntax-cdr))
(syntax . .(make-binding 'core exp-syntax))
(list . .(make-binding 'core 'list)))
(let ([label]* (map (lambda (x) (make-label)) id-binding*)))
(values
`(.,@(map (lambda (sym label)
  (make-subst sym (list top-mark) label))
  (map car id-binding*)
  label*)
.top-mark)
(map cons label)* (map cdr id-binding*)))))

```

В добавление к перечисленным входным данным начальная среда также должна включать связывания для встроенных синтаксических выражений, которые мы не внедряли (например letrec и let-syntax), а также для всех встроенных процедур Scheme. Она также должна включать полную версию syntax, а вместо syntax-pair?, syntax-car и syntax-cdr она должна включать syntax-case.

Пример

Пройдемся теперь по следующему примеру, взятому из начала главы:

```
(let ([t #'t]) (or #f t))
```

Предположим, что выражение or было определено для выполнения преобразования, заданного в начале этой главы, с использованием эквивалента следующего определения or из раздела «Краткое введение в syntax-case»:

```

(define-syntax or
  (lambda (x)
    (syntax-case x ()
      [(_ e1 e2) #'(let ([t e1]) (if t t e2))]))

```

Вначале расширителю предоставляется объект синтаксиса, чье выражение имеет вид (let ([t #'t]) (or #f t)), а оболочка пуста, за исключением содержимого начальной оболочки, которое мы для краткости опускаем. (Мы идентифицируем объекты синтаксиса путем помещения выражения и входных данных оболочки, если таковые имеются, в угловые скобки.)

```
<(let ((t #'t)) (or #f t)>
```

Расширителю предоставляется также начальная среда, которая, как мы предполагаем, содержит связывание для макрода or, а также для основного выражения и встроенных процедур. Эти входные данные среды также опущены

для краткости, наряду с метасредой, которая здесь не играет никакой роли, поскольку мы не расширяем никаких преобразуемых выражений.

Выражение `let` распознается как основное, поскольку `let` присутствует в начальной оболочке и среде. Преобразователь для `let` рекурсивно расширяет правую часть выражения `#t` во входной среде, выдавая в результате `#t`. Он также рекурсивно расширяет тело с расширенной оболочкой, которая отображает `x` на новую метку `11`:

```
<(or #f t) [t x () → 11]>
```

Подстановки, показанные в скобках, имя и список маркеров, отделены символом `x`, а метка следует за стрелкой вправо.

Среда также расширяется, чтобы отобразить метку на связывание типа `lexical` с новым именем `t.1`:

```
11 → lexical(t.1)
```

Выражение `or` распознается как макровызов, поэтому вызывается преобразователь для `or`, создается новое выражение для вычисления в той же самой среде. Входные данные преобразователя `or` помечены новым маркером `m2`, и такой же маркер добавлен к выходным данным, что приводит к следующему:

```
<(<let> ((<t> #f))
  <(if> <t> <t> <t m2 [t x () → 11]>))
m2>
```

Разница между синтаксическими объектами, представляющими введенный идентификатор `t`, и идентификатором `t`, извлеченным из входных данных, имеет важное значение для определения, как каждый из них будет переименован, когда расширитель дойдет до них, и мы накоротке рассмотрим этот момент.

Выражение `#f`, появляющееся в правой части выражения `let`, формально является объектом синтаксиса с той же самой оболочкой, что имеет место для `t`, извлеченного из входных данных, но оболочка для неизменяемой части не имеет значения, поэтому для простоты мы рассматриваем его как не имеющую оболочку.

У нас есть и другое основное выражение `let`. В процессе распознавания и анализа выражения `let` маркер `m2` внедряется в подвыражения:

```
<(let m2> ((<t m2> #f)
  <(<if> <t> <t> <t m2 [t x () → 11]>)
  m2)>
```

Преобразователь для `let` рекурсивно расширяет правую часть выражения `#f`, выдавая в результате `#f`, затем рекурсивно расширяет тело с расширением оболочки, отображая введенный идентификатор `t` с маркером `m2` на новую метку `12`:

```
<(<if> <t> <t> <t m2 [t x () → 11]>
  [t x (m2) → 12]
  m2>
```

Среда также расширяется, чтобы отобразить метку на связывание типа `lexical` с новым именем `t.2`:

```
12 → lexical(t.2). 11 → lexical(t.1)
```

Получающееся в результате этого выражение распознается как основное выражение `if`. В процессе его распознавания и анализа расширитель внедряет

внешние подстановки и маркеры в составные части. Маркер $m2$, который уже фигурировал в оболочке для последнего появления t , отменяет маркер $m2$ во внешней оболочке, оставляя это появление t без маркировки:

```
<if [t × (m2) → 12] m2>
<t [t × (m2) → 12] m2>
<t [t × (m2) → 12] m2>
<t [t × (m2) → 12] [t × () → 11]>
```

Преобразователь для if рекурсивно обрабатывает его подвыражения во входной среде. Первое подвыражение:

```
<t [t × (m2) → 12] m2>
```

распознается как ссылка на идентификатор, поскольку выражение является символом (t). Подстановка, фигурирующая в оболочке, в данном случае применяется, поскольку имя (t) и маркировки ($m2$) одинаковы. Поэтому расширитель ищет 12 в среде и обнаруживает, что она отображается на лексическую переменную $t.2$. Второе подвыражение точно такое же и поэтому тоже отображается на $t.2$. А вот третье подвыражение отличается:

```
<t [t × (m2) → 12] [t × () → 11]>
```

В этом идентификаторе отсутствует маркер $m2$, поэтому первая подстановка не применяется, даже при том, что имя совпадает. А вторая подстановка применяется, поскольку у нее точно такое же имя и такой же набор маркеров (ни один из которых не принадлежит к маркеру вершины из вскрытой начальной оболочки). Поэтому расширитель ищет 11 в среде и обнаруживает, что он отображается на $t.1$.

На выходе выражение if реконструируется следующим образом:

```
(if t.2 t.2 t.1)
```

Внутреннее выражение let реконструируется следующим образом:

```
(let ([t.2 #f]) (if t.2 t.2 t.1))
```

И внешнее выражение let реконструируется следующим образом:

```
(let ([t.1 #t]) (let ([t.2 #f]) (if t.2 t.2 t.1)))
```

что в точности соответствует желаемому, хотя специальный выбор новых имен не столь важен, пока они отличаются друг от друга.

Вывод

Рассмотренный в этой главе упрощенный расширитель иллюстрирует основной алгоритм, который лежит в основе полноценной реализации *syntax-case*, без тех сложностей, которые относятся к механизму соответствия образцу, обработке внутренних определений и дополнительных основных выражений, которые обычно обрабатываются расширителем. Представление среды приспособлено под имеющие единственное связывание выражения *lambda*, *let* и *letrec-syntax*, реализованные расширителем; обычно на практике используется более эффективная реализация, справляющаяся с группами связываний. Хотя эти до-

полнительные свойства не просты для добавления, концептуально они независимы от алгоритма расширения.

Расширитель syntax-case кроме всего прочего раздвигает границы алгоритма гигиенического макрорасширения KFFD за счет поддержки локальных синтаксических связываний и управляемого захвата, а также исключает растущие в геометрической прогрессии издержки, присущие алгоритму KFFD.

Алгоритм KFFD прост и элегантен, и основанный на нем расширитель, безусловно, может считаться примером красивого программного кода. С другой стороны, расширитель syntax-case в силу объективных причин значительно сложнее. Но он не менее красив, поскольку красота может просматриваться и в сложных программах, пока они сохраняют хорошую структуру и успешно справляются с теми задачами, для которых создавались.

26 Архитектура, экономящая силы: объектно-ориентиро- ванная рабочая среда для программ с сетевой структурой

Уильям Отт (William R. Otte) и Дуглас Шмидт (Douglas C. Schmidt)

Разрабатывать программы для сетевых приложений непросто, а создавать такие программы для многократного использования еще сложнее. В первую очередь, из-за трудностей, присущих системам их распространения, в числе которых оптимальное соответствие служб приложения аппаратным узлам, синхронизация инициализации службы и обеспечение работоспособности при маскировке частичных отказов. Все эти трудности способны загнать в тупик даже опытных разработчиков программного обеспечения, поскольку возникают на почве фундаментальных проблем, имеющихся в сфере сетевого программирования.

К сожалению, разработчикам нужно также справляться и с дополнительными трудностями, в числе которых низкоуровневые и непереносимые интерфейсы программ и использование функционально-ориентированных конструкторских технологий, требующих утомительных и не гарантированных от ошибок переработок по мере развития требований и (или) платформ. В большинстве случаев эти трудности возникают из-за ограничений, присущих инструментальным средствам и технологиям, используемым в силу исторически сложившихся причин разработчиками сетевого программного обеспечения.

Несмотря на то что объектно-ориентированные технологии используются во многих областях, таких как графические пользовательские интерфейсы и эффективные инструментальные средства, в большей части сетевого программного обеспечения до сих пор используются интерфейсы прикладного программирования (API) операционных систем (OS), созданные на языке Си, такие как Unix socket API или поточный интерфейс прикладного программирования Windows. Множество дополнительных трудностей сетевого программирования возникает из-за использования этих OS API на уровне языка Си, в которых не обеспечивается безопасность типов, зачастую становится невозможным повторное использование и переносимость между платформами операционных

систем. Интерфейсы прикладного программирования на языке Си проектировались еще до того, как современные методы проектирования и технологии получили широкое распространение, поэтому они заставляют разработчиков раскладывать их задачи по функциям, пользуясь понятиями технологических операций, проектируемых сверху вниз, вместо того чтобы использовать объектно-ориентированные подходы к проектированию и технологиям программирования. Опыт последних нескольких десятилетий показал, что функциональная декомпозиция нетривиального программного обеспечения усложняет обслуживание и развитие, поскольку функциональные требования вряд ли можно считать стабильными основами проектирования¹.

К счастью, проводившееся в течение двух десятилетий совершенствование технологий проектирования и реализации, а также языков программирования существенно облегчило написание и повторное использование сетевого программного обеспечения. В частности, объектно-ориентированные языки программирования (такие как C++, Java и C#) в сочетании с *паттернами* (такими как Фасады оболочек (Wrapper Facades)², Адаптеры (Adapters) и Шаблонный метод (Template Method³)), и рабочими средами (такими как связующее программное обеспечение узлов вроде ACE⁴ и библиотеки классов языка Java для сетевого программирования) помогают инкапсулировать низкоуровневые функциональные интерфейсы OS API и скрыть синтаксические и семантические различия между платформами. Благодаря этому разработчики могут сосредоточиться на поведении и свойствах своего программного обеспечения, которые отражают специфику его применения, вместо того чтобы каждый раз сражаться с дополнительными трудностями низкоуровневого сетевого программирования и инфраструктуры OS.

Основная польза от применения паттернов и рабочих сред к сетевому программному обеспечению состоит в том, что они в состоянии помочь разработчикам создавать многократно используемые архитектуры, которые (1) фиксируют общую структуру и поведение в определенной области и (2) облегчают выборочный обмен или замену различных алгоритмов, стратегий и механизмов без негативного влияния на другие имеющиеся компоненты архитектуры. Наряду с тем что большинство разработчиков сетевого программного обеспечения могут для своих приложений использовать хорошо отработанные объектно-ориентированные рабочие среды, сведения о том, как их создавать, по-прежнему пребывают в области «черной магии» и по исторически сложившимся традициям приобретаются только за счет довольно масштабного (и дорогостоящего) применения метода проб и ошибок.

¹ Bertrand Meyer, «Object-Oriented Software Construction», Second Edition, Prentice Hall, 1997.

² Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, «Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects», John Wiley and Sons, 2000.

³ Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995.

⁴ Douglas C. Schmidt and Stephen D. Huston, «C++ Network Programming, Vol. 2: Systematic Reuse with ACE and Frameworks», Addison-Wesley Longman, 2003.

В дополнение к обычным непростым задачам разработки гибких объектно-ориентированных конструкций, обладающих возможностями по расширению и сулящих поддержку новых требований, сетевое программное обеспечение зачастую должно эффективно работать и расширяться в целом ряде операционных сред. Задача этой главы — убрать покров таинственности с объектно-ориентированных рабочих сред для сетевых программ путем изучения конкретного примера с целью проведения системного анализа конструкции программы, относящейся к типовому сетевому приложению.

В общих чертах красота нашего решения состоит в применении паттернов и объектно-ориентированных технологий с целью выдержать ключевой баланс интересов в этой области, включая многократность использования, расширяемость и производительность. В частности, наш подход позволяет разработчикам вскрыть типовые факторы, мешающие проектированию и программированию, и повысить тем самым многократность использования программ. Он также предоставляет средства для инкапсуляции переменных общим и параметризируемым способом, повышая тем самым расширяемость и возможность переноса программных средств.

Типовое приложение: служба регистрации

Объектно-ориентированная программа, используемая в качестве основного компонента нашего учебного примера, представляет собой сетевую службу регистрации. Как показано на рис. 26.1, эта служба состоит из клиентских приложений, которые генерируют записи регистрационного журнала и посылают их на центральный сервер регистрации, который принимает и сохраняет регистрационные записи для их дальнейшего просмотра и обработки.

Компонент, представляющий собой регистрационный сервер нашей сетевой системы регистрации (на рис. 26.1 он находится в центре), дает нам идеальную среду для демонстрации красоты объектно-ориентированной сетевой программы, поскольку он показывает следующий диапазон возможных вариантов, из которых разработчики могут выбирать конструктивные особенности созданного сервера.

- Различные механизмы взаимодействия между процессами (*interprocess communication* — IPC) (среди которых сокеты, SSL, использование общего пространства памяти, TLI, поименованные каналы и т. д.), которые разработчики могут использовать для отправки и получения регистрационных записей.
- Различные модели параллельной обработки (среди которых итеративные, реактивные, с отдельным потоком на каждое соединение, с отдельным процессом на каждое соединение, с использованием различных типов пулов потоков и т. д.), которые разработчики могут использовать для обработки регистрационных записей.
- Различные стратегии блокировки (среди которых блокировки на уровне потока или рекурсивные мьютексы на уровне процесса, нерекурсивные мьютексы, блокировки чтения-записи, нулевые мьютексы и т. д.), которые разработчики могут использовать для обеспечения последовательного доступа

к таким ресурсам, как счетчик количества запросов, используемый совместно несколькими потоками.

- Различные форматы регистрационных записей, которые могут передаваться от клиента к серверу. Когда сервер получает регистрационные записи, они могут обрабатываться различными способами, например, выводиться на консоль, сохраняться в едином файле или даже в отдельном файле для каждого клиента, для того чтобы максимально увеличить число параллельных записей на диск.

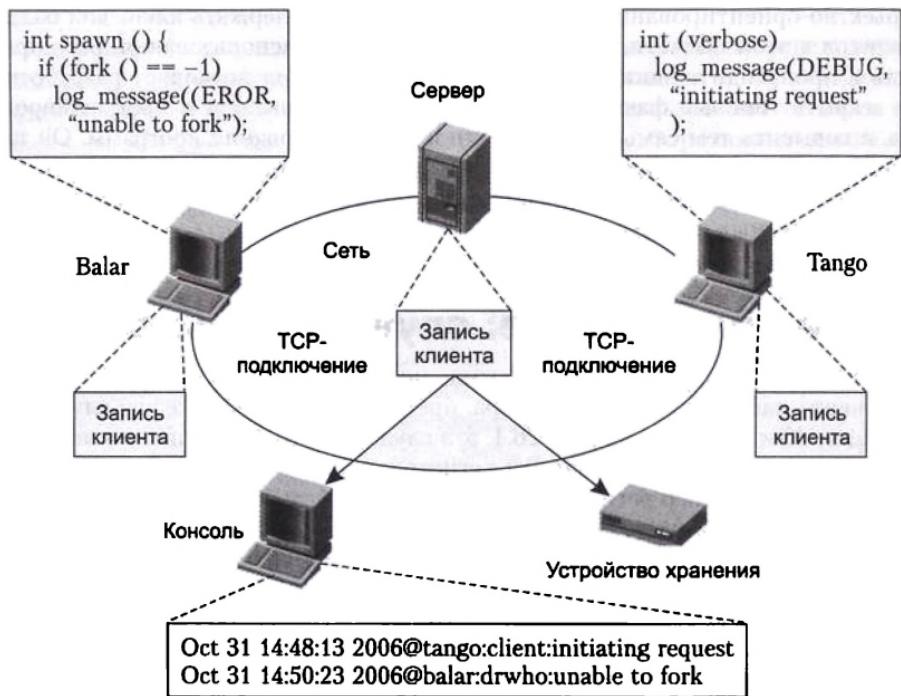


Рис. 26.1. Архитектура сетевой службы регистрации

Реализация любой из этих комбинаций, скажем, запуска по одному потоку на подключение, использования на регистрационном сервере IPC на основе сокета и нерекурсивных мьютексов на уровне потока, представляет собой относительно простую задачу. Но решение «все-в-одном» не может отвечать потребностям всех регистрируемых служб, поскольку различия в клиентских требованиях и разные операционные среды могут оказывать очень разное влияние на компромиссы времени выполнения и отводимого пространства памяти, стоимость проекта и сроки его разработки. Поэтому ключевой задачей является разработка конфигурируемого сервера регистрации, способного *свободно расширяться*, чтобы отвечать новым потребностям и требовать для этого приложения *минимума усилий*.

В центре решения этой задачи лежит всестороннее понимание паттернов и связанных с ними технологий проектирования, необходимых для разработки объектно-ориентированных сред, способных эффективно справляться:

- со сбором общей структуры и поведения в базовых и родовых классах;
- с допуском избирательной настройки поведения с помощью подклассов и за счет предоставления определенных параметров базовым классам.

На рис. 26.2 показана конструкция объектно-ориентированной среды сервера регистрации, отвечающая этим целям. Ядром этой конструкции является класс `Logging_Server`, определяющий общую структуру и функциональные возможности для сервера регистрации посредством использования следующих элементов:

- параметризованных типов C++, позволяющих разработчикам откладывать выбор типов данных, используемых в родовых классах или функциях до момента их реализации;
- паттерна Шаблонный метод (Template Method), который определяет структуру алгоритма, поручая отдельные этапы методам, которые могут быть заменены при помощи подклассов;
- паттерна Фасад оболочки (Wrapper Facade), который инкапсулирует интерфейсы API, не имеющие объектной ориентации, и данные внутри обеспечивающих типовую безопасность объектно-ориентированных классов.

Подклассы и конкретная реализация `Logging_Server` улучшают эту всеобщую многократно используемую архитектуру, чтобы настроить изменяемые шаги поведения сервера регистрации за счет выбора требуемых IPC-механизмов, моделей параллельной обработки и стратегий блокировки. Таким образом, `Logging_Server` имеет *архитектуру для ряда продуктов*¹, которая определяет комплексный набор классов, совместно работающих, чтобы определить многократно используемую конструкцию для семейства родственных серверов регистрации.

Весь остальной материал главы построен так: в следующем разделе дается описание объектно-ориентированного проектирования рабочей среды сервера регистрации, исследуется ее архитектура, рассматриваются все факторы, повлиявшие на конструкцию объектно-ориентированной среды, чтобы показать, почему мы выбрали те или иные паттерны и свойства языка, а также резюмируются альтернативные подходы, отвергнутые нами по тем или иным причинам. В двух следующих разделах представлено несколько последовательных программных реализаций рабочей среды сервера регистрации на C++ и программные реализации этой среды, рассчитанные на параллельную обработку данных. В завершение резюмируются все достоинства объектно-ориентированной концепции и тех технологий создания программного обеспечения, которые рассмотрены в этой главе.

¹ Paul Clements and Linda Northrop, «Software Product Lines: Practices and Patterns», Addison-Wesley, 2001.

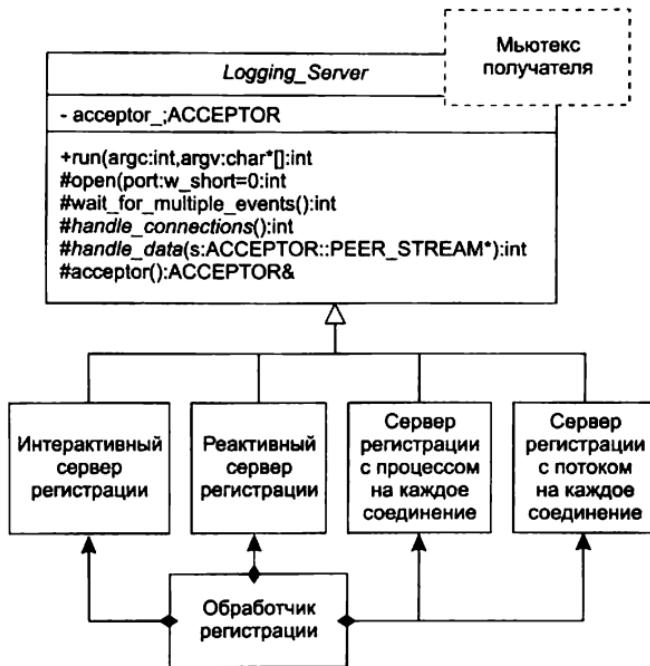


Рис. 26.2. Объектно-ориентированное проектирование рабочей среды сервера регистрации

Объектно-ориентированное проектирование рабочей среды сервера регистрации

Прежде чем приступить к обсуждению объектно-ориентированного проектирования нашего сервера регистрации, важно усвоить несколько ключевых понятий, касающихся объектно-ориентированных рабочих сред. Многие программисты знакомы с концепцией библиотеки классов, которая представляет собой набор многократно используемых классов, обеспечивающих функциональные возможности, которыми можно воспользоваться при разработке объектно-ориентированных программ. Объектно-ориентированная рабочая среда расширяет преимущества использования библиотек классов за счет следующих моментов¹:

Она определяет «полуфабрикаты» приложений, которые включают в себя структуру объектов и функциональные возможности, относящиеся к определенной области задач

¹ Ralph Johnson, «Frameworks – Patterns + Components», Communications of the ACM, vol. 40, № 10, Oct. 1997.

Классы в рабочих средах ведут совместную работу по обеспечению общей архитектурной основы в определенной области, такой как графические пользовательские интерфейсы, прокладка курсов воздушных судов или сетевые службы регистрации. Законченные приложения могут быть скомпонованы за счет наследования и (или) создания экземпляров компонентов среды. В отличие от них, библиотеки классов не имеют столь характерной направленности на применение в определенных областях и не дают столь же широкой сферы для повторного применения. Например, такие компоненты библиотеки классов, как классы для обработки строк, комплексных чисел, массивов и побитовой обработки, считаются относительно низкоуровневыми и используются повсеместно во многих прикладных областях.

Рабочая среда проявляет активность и демонстрирует «инверсию управления» в процессе работы

Библиотекам классов свойственна пассивность — то есть они выполняют изолированные части общей работы, когда вызываются управляющими потоками из самоуправляемых объектов приложения. В отличие от этого, рабочая среда является активной, то есть она направляет управляющие потоки внутри приложения, используя для этого паттерны диспетчеризации событий, такие как Reactor¹ и Observer². «Инверсия управления» в архитектуре рабочей среды в процессе работы часто сравнивается с «Принципом Голливуда», который звучит так: «Не звоните нам, мы вам сами позвоним»³.

Рабочая среда обычно проектируется на основе анализа различных потенциальных проблем, на решение которых она может быть нацелена, и определения, какие части каждого решения идентичны, а какие — уникальны. Этот метод проектирования называется *анализом общих и изменяемых свойств*⁴ и охватывает следующие темы.

Сфера применения

Определение поля деятельности (то есть проблемных областей, на решение которых нацелена рабочая среда) и контекста рабочей среды.

Общие свойства

Описание свойств, повторяющихся во всех представителях семейства продуктов, основанных на данной рабочей среде.

Изменяемые свойства

Описание свойств, уникальных для различных представителей семейства продуктов.

¹ Schmidt и др., *указ. соч.*

² Gamma и др., *указ. соч.*

³ John Vlissides, «Pattern Hatching – Protection, Part I: The Hollywood Principle», C++ Report, Feb. 1996.

⁴ J. Coplien, D. Hoffman, and D. Weiss, «Commonality and Variability in Software Engineering». IEEE Software, vol. 15, № 6, Nov.-Dec. 1998.

Определение общих свойств

В силу изложенного первое, что мы должны сделать при проектировании рабочей среды сервера регистрации, — это понять, какие части системы могут быть реализованы рабочей средой (могут стать общими), а какие — останутся специализированными в подклассах или параметрах (будут изменяемыми). Этот анализ провести несложно, поскольку процессы, задействованные в обработке регистрационной записи, отправляемой по сети, могут быть разбиты на шаги, показанные на рис. 26.3, которые имеют общий характер для всех реализаций сервера регистрации.

На этой стадии процесса проектирования мы определяем каждый шаг как можно абстрактнее. К примеру, на этой стадии мы выстраиваем минимум предположений о типе IPC-механизмов, кроме случаев, когда они являются ориентированными на подключение для обеспечения надежности доставки регистрационных записей.

Более того, мы не будем определять тип стратегии параллельной обработки (то есть сможет ли сервер обрабатывать несколько запросов, и если да, то как он с ними справится) или механизм синхронизации, используемый при каждом шаге. Таким образом, текущий выбор специфики поведения для шага сведен к последовательной конкретной реализации, обеспечивающей вполне определенный вариант поведения для каждого шага.

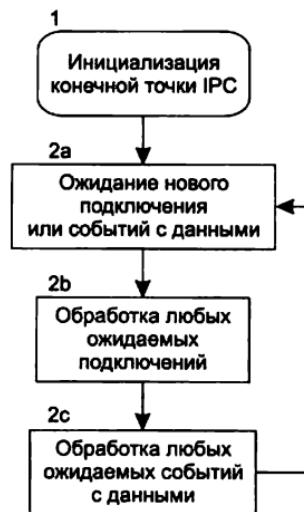


Рис. 26.3. Основной цикл сервера регистрации

Удобным способом определения абстрактных шагов и отсрочки реализации специфики их поведения на более поздние этапы процесса проектирования является использование паттерна Шаблонный метод. Этот паттерн определяет базовый класс, реализующий общие свойства, но абстрактно — шагами в *шаблонном методе* в понятиях *методов-ловушек* (*hook methods*), которые могут

быть выборочно заменены конкретными реализациями. Чтобы обеспечить определение всех конкретных реализаций через методы-ловушки, должны использоваться такие свойства языков программирования, как «чистые» виртуальные функции в C++ или абстрактные методы в Java.

На рис. 26.4 показана структура паттерна Шаблонный метод и продемонстрировано, как этот паттерн применяется к проектированию нашей объектно-ориентированной среды сервера регистраций (Logging Server).

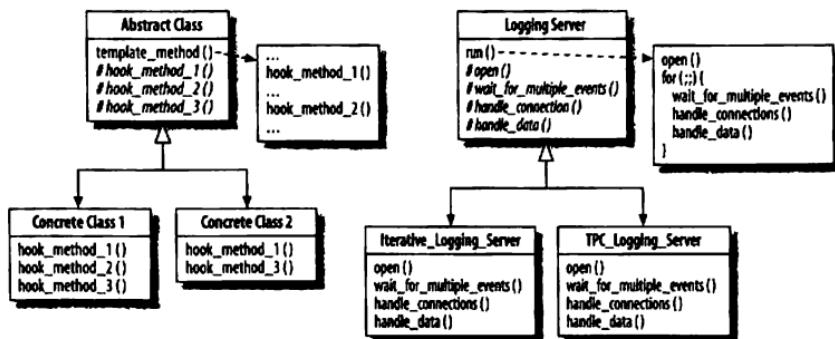


Рис. 26.4. Паттерн Шаблонный метод и его применение к серверу регистрации

Увязка вариантов

Хотя использование паттерна Шаблонный метод направлено на полнообъемное проектирование шагов нашей рабочей среды сервера регистрации, остается открытый вопрос, как увязать все три определенные ранее измерения изменчивости (IPC, параллельная обработка и механизмы синхронизации), необходимые для обеспечения нашего проекта. Можно, конечно, просто воспользоваться паттерном Шаблонный метод и реализовать по одной комбинации IPC-параллельная обработка-синхронизация в каждом конкретном подклассе. К сожалению, такой подход приведет к росту количества конкретных подклассов в геометрической прогрессии, поскольку каждое дополнение к любому из измерений может порождать новую реализацию для каждой возможной комбинации других измерений. Поэтому использование проектирования на основе Шаблонного метода в чистом виде не будет иметь существенных преимуществ перед созданием отдельных реализаций сервера регистрации для каждого варианта с применением обычного способа.

При более эффективном и масштабируемом способе проектирования можно предусмотреть использование того факта, что наши измерения изменчивости в значительной степени независимы друг от друга. К примеру, выбор различных механизмов IPC вряд ли потребует внесения каких-либо изменений в используемые механизмы параллельной обработки данных или синхронизации. Кроме того, на высшем уровне есть ряд общих свойств в функционировании различных типов IPC и механизмов синхронизации – то есть механизмы IPC

могут инициировать или воспринимать подключения и отправлять или получать данные, а механизмы синхронизации содержат операции для захвата и освобождения блокировок. Задача проектирования состоит в инкапсуляции дополнительных сложностей, присущих этим API, чтобы их можно было использовать взаимозаменяямо.

Решение этой проблемы состоит в использовании паттерна Фасад оболочки (Wrapper Facade), который предоставляет единый унифицированный объектно-ориентированный интерфейс для базового не объектно-ориентированного IPC и механизмов синхронизации, предоставляемых системными функциями OS. Создание фасадов оболочек особенно полезно для улучшения переносимости за счет скрытия дополнительных сложностей среди механизмов, а также за счет упрощения работы и снижения возможностей допущения ошибок при использовании этих API. К примеру, фасад оболочки может определить высокуюровневую систему типов, обеспечивающую вызов в базовых не объектно-ориентированных (и менее безопасных по типам) OS IPC и структурах данных синхронизации только корректных операций. Роль фасада оболочки показана на рис. 26.5.

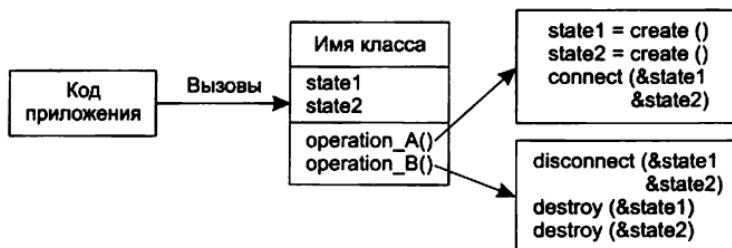


Рис. 26.5. Паттерн проектирования Фасад оболочки

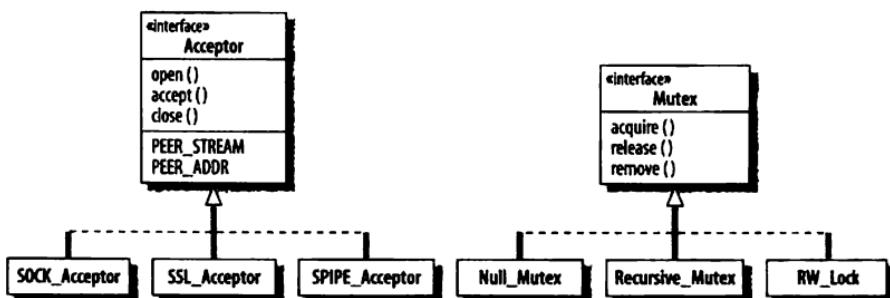


Рис. 26.6. Некоторые имеющиеся в ACE фасады оболочек для создания пассивного подключения и синхронизации

Примером связующего программного обеспечения хост-инфраструктуры является библиотека ACE, получившая широкое распространение и определяющая унифицированные объектно-ориентированные интерфейсы, использующие фасады оболочек как для IPC, так и для механизмов синхронизации. За ос-

нову фасадов оболочек, используемых в этой главе, мы взяли их упрощенные версии, предоставляемые библиотекой ACE. На рис. 26.6 показаны некоторые имеющиеся в ACE фасады оболочек.

Фасад оболочки Acceptor дает средства для создания подключений в пассивном режиме и обеспечивает «особенности» для представления аспектов механизма, работающего практически одинаково для различных реализаций, созданных с использованием различных API. Например, PEER_STREAM и PEER_ADDR обозначают, соответственно, подчиненные фасады оболочек для отправки/получения данных и для адресации с помощью механизма IPC. SOCK_Acceptor является подклассом Acceptor, который используется в этой главе для создания фабрики пассивной установки подключения, реализуемого с помощью socket API.

Фасад оболочки Mutex обеспечивает интерфейс, методы которого захватывают и освобождают блокировки, включая Recursive_Mutex, реализованный с использованием мьютекса, который не будет вызывать взаимную блокировку при многократном захвате одним и тем же потоком, RW_Lock, реализующий семантику блоков чтения-записи, и Null_Mutex, чьи методы acquire()/release() действуют как пустые операции. Последний класс упоминается в качестве примера паттерна несуществующего объекта – Null Object¹, и он применяется для ликвидации синхронизации без изменения кода приложения. При просмотре рис. 26.6 создается впечатление, будто каждое семейство классов связано наследованием, но на самом деле они реализованы за счет классов, не связанных наследованием, у которых имеется общий интерфейс и которые могут использоваться как параметры типов для шаблонов C++. Мы выбрали такую конструкцию, чтобы избежать издержек вызова виртуального метода.

Связывание воедино

Другая сложная задача проектирования – найти способ связывания стратегии параллельной обработки с IPC и с механизмом синхронизации. Для этого можно применить паттерн Стратегии (Strategy)², который инкапсулирует алгоритмы в виде объектов, чтобы они могли заменяться в процессе выполнения. Этот подход обеспечит Logging_Server указателем на абстрактные базовые классы, имеющиеся в Acceptor и Mutex, а затем, в зависимости от динамического связывания и полиморфизма, позволит направлять виртуальные методы на соответствующие экземпляры подклассов.

Хотя осуществление подхода, основанного на использовании Стратегии, вполне реально, это решение нельзя признать идеальным. Каждая входящая регистрационная запись должна генерировать несколько вызовов методов в фасадах оболочек Acceptor и Mutex. Но из-за этого может снизиться производительность, поскольку вызовы виртуальных методов приносят более значительные издержки, чем вызовы невиртуальных методов. С учетом того что динамическая

¹ Bobby Woolf, «The Null Object Pattern», в Robert C. Martin, Dirk Riehle, and Frank Buschmann, «Pattern Languages of Program Design, Volume 3», Addison-Wesley, 1997.

² Gamma и др., указ. соч.

замена механизмов IPC или синхронизации не входит в круг требований для наших серверов регистраций, более эффективным решением будет использование параметризованных типов C++ для реализации наших классов сервера регистрации с фасадами оболочек для IPC и синхронизации.

Поэтому мы определяем следующий универсальный абстрактный класс под названием `Logging_Server`, из которого будут наследоваться все серверы регистрации этой главы:

```
template <typename ACCEPTOR, typename MUTEX>
class Logging_Server {
public:
    typedef Log_Handler<typename ACCEPTOR::PEER_STREAM> HANDLER;

    Logging_Server (int argc, const char *argv):
        // Шаблонный метод, запускающий каждый шаг в основном событийном цикле.
        virtual void run (void);

protected:
    // Методы-ловушки, позволяющие иметь вариации каждого шага.
    virtual void open (void);
    virtual void wait_for_multiple_events (void) = 0;
    virtual void handle_connections (void) = 0;
    virtual void handle_data
        (typename ACCEPTOR::PEER_STREAM *stream = 0) = 0;

    // Приращение счетчика запроса, защищенного мьютексом.
    virtual void count_request (size_t number = 1);

    // Экземпляр параметра шаблона, принимающего подключения.
    ACCEPTOR acceptor_;

    // Хранилище количества полученных регистрационных записей.
    size_t request_count_;

    // Экземпляр параметра шаблона, который организует последовательный доступ
    // к request_count_.
    MUXEX mutex_;

    // Адрес, который будет отслеживаться сервером для подключений.
    std::string server_address_;
};


```

Большинство методов в `Logging_Server` являются чисто виртуальными, что обеспечивает их выполнение подклассами. Но следующие методы `open()` и `count_request()` многократно используются всеми серверами регистрации, рассматриваемыми в этой главе:

```
template <typename ACCEPTOR, typename MUXEX>
Logging_Server<ACCEPTOR, MUXEX>::Logging_Server
(int argc, char *argv[]): request_count_(0) {
    // Анализ аргументов argv и сохранение серверного address_...
}

template <typename ACCEPTOR, typename MUXEX> void
```

```

Logging_Server<ACCEPTOR, MUTEX>::open (void) {
    return acceptor_.open (server_address_);
}

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::count_request (size_t number) {
    mutex_.acquire (); request_count_ += number; mutex_.release ();
}

```

Класс Log_Handler отвечает за демаршализацию регистрационной записи из потока данных подключения, чей IPC-механизм определяется типом параметра ACCEPTOR. Реализация этого класса лежит за пределами тематики этой главы и может быть еще одним измерением изменчивости – то есть серверы регистрации могут потребовать поддержки различных форматов регистрационных сообщений. Если бы нам нужно было поддерживать различные форматы или методы хранения входящих регистрационных сообщений, этот класс мог бы стать еще одним параметром нашей регистрирующей рабочей среды.

Для достижения наших целей достаточно знать, что он параметризируется IPC-механизмом и предоставляет два метода: peer(), который возвращает ссылку на поток данных, и log_record(), который читает отдельную регистрационную запись из потока.

Основная точка входа в Logging_Server – это шаблонный метод под названием run(), который реализует шаги, намеченные на рис. 26.3, делегируя определенные шаги методам ловушек, объявленным в защищенном разделе Logging_Server, как показано в следующем кодовом фрагменте:

```

template <typename ACCEPTOR, typename MUTEX> void
Logging_Server<ACCEPTOR, MUTEX>::run (void) {
    try {
        // Шаг 1: инициализация конечной точки IPC-фабрики для отслеживания
        // новых подключений по адресу сервера. ///
        open ( );
    }

        // Шаг 2: Вход в событийный цикл
        for (;;) {
            // Шаг 2a: ожидание новых подключений или поступления
            // новых регистрационных записей.
            wait_for_multiple_events ( );
            // Шаг 2b: восприятие нового подключения (если оно доступно)
            handle_connections ( );

            // Шаг 2c: обработка полученной регистрационной записи (если она доступна)
            handle_data ( );
        }
    } catch (...) { /* ... Обработка исключения ... */ }
}

```

Красота этого кода заключается в следующем.

- Паттерны, положенные в основу его конструкции, облегчают обработку изменений в моделях параллельной обработки, таких как изменения поведения шаблонного метода run() за счет предоставления определенных реализаций методов-ловушек в реализации подклассов.

- Его основанная на шаблонах конструкция помогает справиться с вариациями механизмов IPC и синхронизации, подключая, к примеру, различные типы внутри параметров шаблонов ACCEPTOR и MUTEX.

Реализация последовательных серверов регистрации

В этом разделе показана реализация серверов регистрации, для которых характерны последовательные модели взаимозамещаемой обработки — то есть вся обработка проводится в едином потоке. Мы рассмотрим как итеративную, так и реактивную реализацию последовательных серверов регистрации.

Итеративный сервер регистрации

Итеративные серверы обрабатывают все регистрационные записи от каждого клиента, прежде чем обработать любые записи от следующего клиента. Поскольку необходимости порождения или синхронизации потоков для этого не требуется, мы используем фасад Null_Mutex для параметризации шаблона подкласса Iterative_Logging_Server, как показано в следующем примере:

```
template <typename ACCEPTOR>
class Iterative_Logging_Server : 
    virtual Logging_Server<ACCEPTOR, Null_Mutex> {
public:
    typedef Logging_Server<ACCEPTOR, Null_Mutex>::HANDLER HANDLER;
    Iterative_Logging_Server (int argc, char *argv[]);

protected:
    virtual void open (void);
    virtual void wait_for_multiple_events (void) {};
    virtual void handle_connections (void);
    virtual void handle_data
        (typename ACCEPTOR::PEER_STREAM *stream = 0);
    HANDLER log_handler_;

    // Для всех клиентов используется общий регистрационный файл.
    std::ofstream logfile_;
};
```

Реализация этой версии нашего сервера довольно проста. Метод open() вносит характерные особенности в поведение метода из базового класса Logging_Server за счет открытия выходного файла перед передачей полномочий родительскому методу open():

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::open (void) {
    logfile_.open (filename_.c_str ());
    if (!logfile_.good ()) throw std::runtime_error;
    // Передача полномочий родительскому методу open().
    Logging_Server<ACCEPTOR, Null_Mutex>::open ();
}
```

Метод `wait_for_multiple_events()` является пустой операцией. Он не нужен, поскольку мы всякий раз обрабатываем только одно подключение. Поэтому метод `handle_connections()` просто осуществляет блокировку до тех пор, пока не будет установлено новое подключение:

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_connections (void)
{ acceptor_.accept (log_handler_.peer ()); }
```

И наконец, `handle_data()` просто считывает регистрационные записи, полученные от клиента, и записывает их в регистрационный файл, занимаясь всем этим до тех пор, пока клиент не закроет подключение или не произойдет ошибка:

```
template <typename ACCEPTOR> void
Iterative_Logging_Server<ACCEPTOR>::handle_data (void) {
    while (log_handler_.log_record (logfile_))
        count_request (' ');
}
```

Хотя реализация итеративного сервера не представляет особых трудностей, его недостаток в том, что он способен одновременно обслуживать только одного клиента. У второго клиента, пытающегося подключиться, может произойти превышение лимита времени ожидания, пока первый клиент завершит свой запрос.

Реактивный сервер регистрации

Реактивный сервер регистрации сглаживает один из основных недостатков итеративного сервера регистрации, рассмотренного в предыдущем разделе, за счет обработки нескольких клиентских подключений и запросов регистрационных записей через относящиеся к операционной системе распределители каналов синхронных событий, имеющиеся в интерфейсах API, предоставляемых OS, такие как `select()` и `WaitForMultipleObjects()`. Эти API в состоянии отслеживать нескольких клиентов, дождаясь в единственном управляющем потоке наступления событий, связанных с вводом–выводом в группе обработчиков ввода–вывода, а затем чередовать обработку регистрационных записей. Поскольку по своей сути реактивный сервер регистрации все еще остается последовательным, он, как показано на рис. 26.7, является наследником ранее реализованного итеративного сервера.

Класс `Reactive_Logging_Server` заменяет все четыре метода-ловушки, которые он наследует у базового класса `Iterative_Logging_Server`. Его метод-ловушка `open()` вносит характерные особенности в поведение метода базового класса, необходимые для инициализации переменных экземпляра `ACE_Handle_Set`, представляющего собой часть фасадов оболочек, упрощающих использование `select()`:

```
template <typename ACCEPTOR> void
Reactive_Logging_Server<ACCEPTOR>::open ( ) {
    // Делегирование полномочий базовому классу.
    Iterative_Logging_Server<ACCEPTOR>::open ( );
```

```

// Пометка активным обработчика, связанного с получателем (acceptor).
master_set_.set_bit (acceptor_.get_handle ());

// Установка обработчика получателя в разблокированный режим.
acceptor_.enable (NONBLOCK);
}
}

```

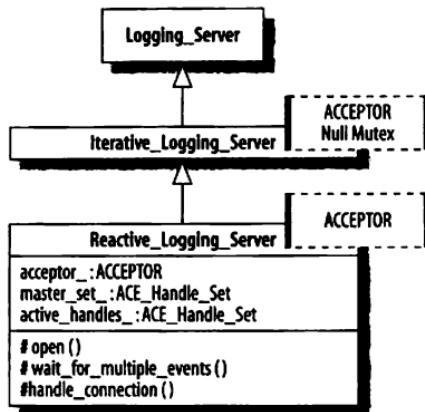


Рис. 26.7. Интерфейс реактивного сервера регистрации

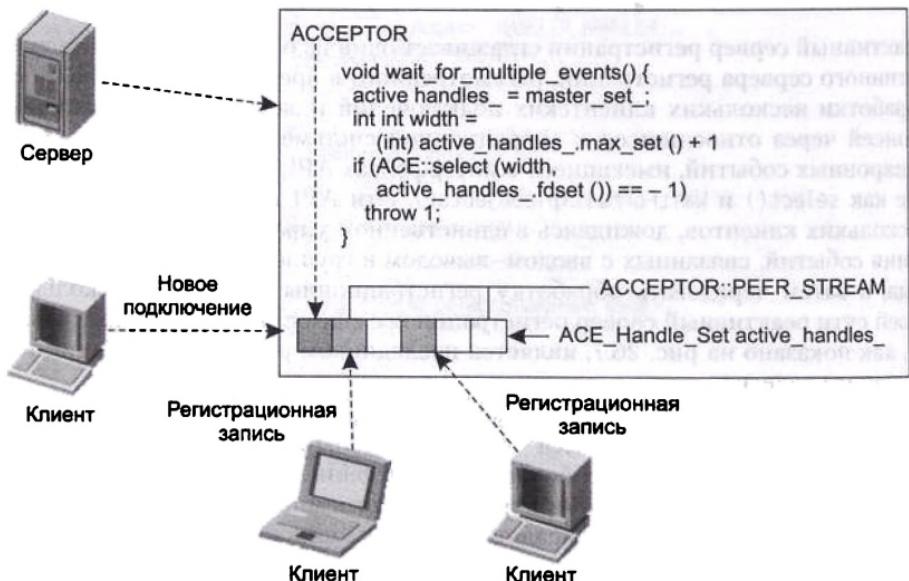


Рис. 26.8. Использование асинхронного распределителя каналов в программе `Reactive_Logging_Server`

В отличие от своей копии в `Iterative_Server`, в этой реализации метод `wait_for_multiple_events()` уже востребован. Как показано на рис. 26.8, этот метод использует распределитель каналов синхронных событий (в данном случае вызов `select()`), чтобы определить, у каких обработчиков ввода–вывода имеется подключение или данные, ожидающие активизации.

После выполнения `wait_for_multiple_events()` у `Reactive_Logging_Server` имеется кэшированный набор обработчиков, ожидающих работу (то есть либо новых запросов на подключение, либо событий получения новых данных), который затем будет обработан другими его методами-ловушками: `handle_data()` и `handle_connections()`. Метод `handle_connections()` проверяет активность обработчика получателей, и если он активен, воспринимает столько подключений, сколько сможет, и кэширует их в `master_handle_set_`. Точно так же метод `handle_data()` осуществляет последовательный перебор оставшихся активных обработчиков, помеченных ранее методом `select()`. Эта работа упрощена за счет использования фасада оболочки ACE socket, которая обеспечивает выполнение экземпляра паттерна Итератор (Iterator)¹ для наборов обработчиков сокетов, как показано на рис. 26.9.

Следующий код реализует основную программу `Reactive_Logging_Server`, использующую `socket API`:

```
int main (int argc, char *argv[]) {
    Reactive_Logging_Server<SOCK_Acceptor> server (argc, argv);
    server.run ();
    return 0;
}
```

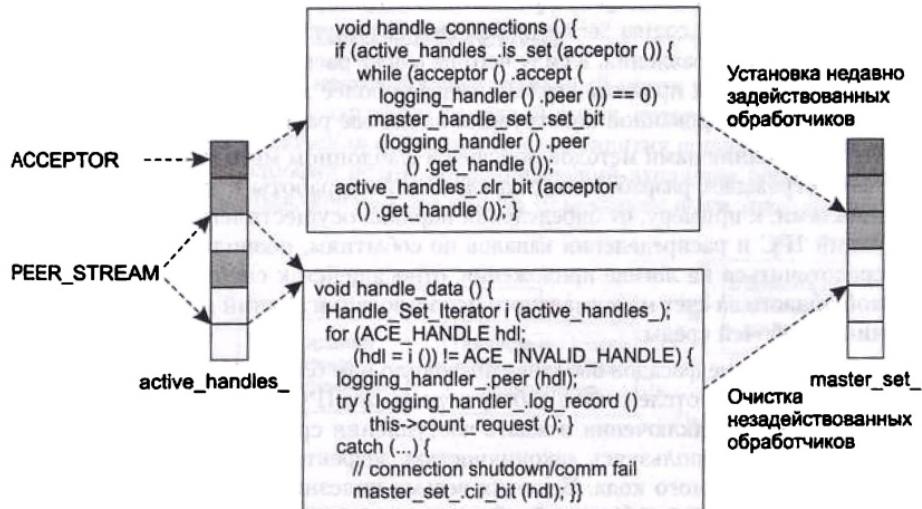


Рис. 26.9. Обработчик событий подключения/данных реактивного сервера

¹ Gamma и др., указ. соч.

Первая строка нашей основной функции параметризирует `Reactive_Logging_Server` типом `SOCK_Acceptor`, что вынудит компилятор C++ сгенерировать код для реактивного сервера регистрации, способного связываться через сокеты. Это в свою очередь приведет к параметризации его базового класса `Logging_Server` с использованием как `SOCK_Acceptor`, так и `Null_Mutex`, посредством жестко заданного аргумента шаблона, предоставленного в процессе наследования. Во второй строке вызывается метод шаблона `run()`, передаваемый базовому классу `Logging_Server`, который сам делегирует полномочия различным методам-ловушкам, реализованным нами в этом классе.

Оценка последовательных решений сервера регистрации

Реактивный сервер регистрации — `Reactive_Logging_Server` улучшил показатели итеративного сервера — `Iterative_Logging_Server` за счет чередующегося обслуживания нескольких клиентов вместо полного обслуживания одного клиента за некий промежуток времени. Но в нем не использованы преимущества имеющегося в операционной системе механизма параллельной обработки данных, поэтому он не может эффективно использовать возможности мультипроцессорной системы. Также он не может сочетать вычисления и работу с подключением за счет обработки регистрационных записей во время чтения новых записей. Эти ограничения являются препятствием для его расширяемости с ростом количества клиентов, даже если используемое аппаратное обеспечение поддерживает одновременное выполнение нескольких потоков.

Хотя `Iterative_Logging_Server` и `Reactive_Logging_Server` запускаются лишь в одном потоке управления, в силу чего не могут расширяться в более продуктивную систему, их простота высвечивает ряд более красивых аспектов нашей объектно-ориентированной конструкции на основе рабочей среды.

- Использование нами методов-ловушек в шаблонном методе `Logging_Server::run()` ограждает разработчиков приложений от работы с низкоуровневыми деталями, к примеру, от определения порядка осуществления сервером операций IPC и распределения каналов по событиям, позволяя тем самым сосредоточиться на логике приложения, относящейся к специфике определенной области за счет максимального использования знаний и опыта разработчиков рабочей среды.
- Использование фасадов оболочек позволило нам блокировать и разблокировать мьютексы, отслеживать работу отдельного IPC-механизма для восприятия нового подключения и ждать наступления сразу нескольких событий ввода–вывода, пользуясь лаконичностью, эффективностью и переносимостью программного кода. Без этих весьма полезных абстракций нам пришлось бы набирать большой объем скучных и не гарантированных от ошибок строк кода, который трудно поддавался бы пониманию, отладке и развитию.

Преимущества этих абстракций становятся еще более ощутимыми при работе с показанными далее более сложными серверами регистрации с параллель-

ной обработкой данных, а также с более сложными примерами использования рабочей среды, в которых используются графический пользовательский интерфейс¹ или коммуникационное связующее программное обеспечение².

Реализация сервера регистрации с параллельной обработкой

Для преодоления ограничений расширяемости итеративных и реактивных серверов, показанных в предыдущих разделах, в серверах регистрации, показанных в этом разделе, используются имеющиеся в операционной системе механизмы параллельной обработки: процессы и потоки. Но использование интерфейсов API, предоставленных операционными системами для порождения потоков или процессов, может стать непростой задачей из-за дополнительных сложностей, имеющихся в их конструкции. Эти сложности возникают из-за семантических и синтаксических различий, существующих не только между различными операционными системами, но и между различными версиями одной и той же операционной системы. Наше решение по преодолению этих сложностей в данном случае заключается в использовании фасадов оболочек, обеспечивающих неизменный интерфейс, не зависящий от применяемой платформы, и объединении этих фасадов оболочек внутри нашей объектно-ориентированной рабочей среды Logging_Server.

Сервер регистрации, использующий отдельный поток на каждое подключение

Наш сервер регистрации, использующий отдельный поток на каждое подключение (TPC_Logging_Server), запускает основной поток, который ожидает и воспринимает новые подключения клиентов. После принятия нового подключения порождается новый рабочий поток, обрабатывающий входящие регистрационные записи от этого подключения. На рис. 26.10 показаны шаги этого процесса.

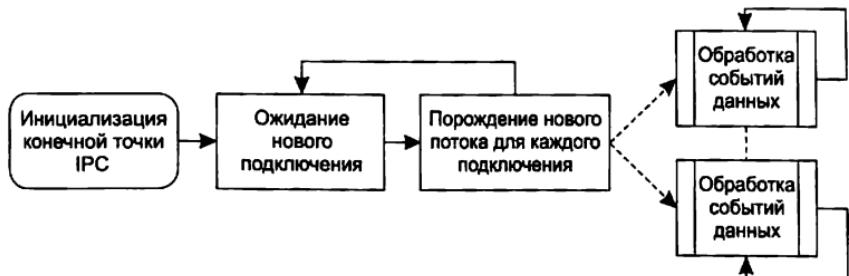


Рис. 26.10. Шаги сервера регистрации, использующего отдельный поток на каждое подключение

¹ Gamma и др., указ. соч.

² Schmidt и др., указ. соч.

Шаги основного цикла для этого сервера регистрации отличаются от шагов, изображенных на рис. 26.3, потому что вызов `handle_data()` уже не нужен, поскольку за него отвечают рабочие потоки. Чтобы справиться с этой ситуацией, есть два пути:

- мы могли заметить, что базовый метод `run()` вызывает `handle_data()`, используя в качестве аргумента по умолчанию указатель `NULL`, и просто добиться от нашей реализации немедленного завершения выполнения для этого входа;
- мы можем просто заменить метод `run()` своей собственной реализацией, игнорирующей этот вызов.

Второе решение на первый взгляд может показаться более подходящим, поскольку при нем удается избежать вызова виртуального метода для `handle_data()`. Но в данном случае первое решение все же лучше, поскольку урон производительности из-за виртуального вызова не является каким-то ограничивающим фактором, а замена шаблонного метода `run()` мешала бы этому классу получать преимущества от изменений в реализации базового класса, создавая предпосылки для его опасного отказа, причины которого будет трудно отследить.

Но главная трудность здесь состоит в реализации самой стратегии параллельной обработки. Как и в варианте `Iterative_Server`, рассмотренном в предыдущем разделе «Итеративный сервер регистрации», метод `wait_for_multiple_events()` является излишним, поскольку наш основной цикл просто ожидает новых подключений, вследствие чего вполне достаточно, чтобы `handle_connections()` был заблокирован на `accept()` и последовательно порождал рабочие потоки для обработки подключившихся клиентов. Поэтому наш класс `TPC_Logging_Server` должен предоставить метод, который будет служить точкой входа для потока. В Си и C++ метод класса может служить точкой входа для потока, только если класс определен как статический, поэтому мы определяем `TPC_Logging_Server::svc()` как метод статического класса.

Теперь нам нужно принять важное конструкторское решение: что именно будет делать точка входа потока? Заманчиво было бы просто заставить сам метод `svc()` выполнять всю работу, необходимую для получения регистрационных записей из связанного с ним подключения. Но эта конструкция далека от идеальной, из-за того что статический метод не может быть виртуальным, поскольку это может вызвать проблемы при последующем получении нового сервера регистрации из этой реализации для изменения способов, которыми он обрабатывает события данных. Тогда разработчики приложений будут вынуждены для вызова правильного статического метода обеспечить полностью идентичную этому классу реализацию `handle_connections()`.

Кроме того, для усиления созданной нами конструкции и кода предпочтительнее логику обработки регистрационной записи иметь внутри метода `handle_data()` и определить вспомогательный объект `Thread_Args`, удерживающий узел сети, возвращенный `accept()`, и указатель на сам объект `Logging_Server`. Поэтому интерфейс нашего класса будет выглядеть в соответствии со схемой, изображенной на рис. 26.11.

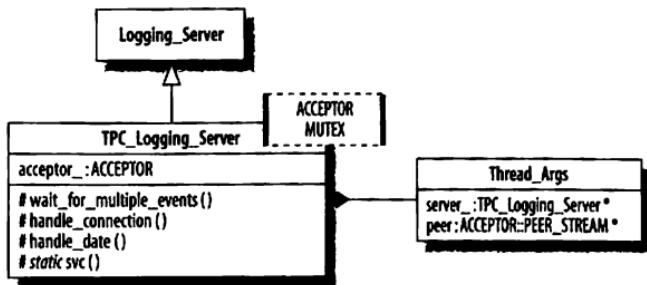


Рис. 26.11. Интерфейс сервера, использующего отдельный поток для каждого подключения

Реализация остальной части **TPC_Logging_Server** трудности не представляет, нужно лишь, чтобы точка входа нашего потока передавала обработку виртуальному методу `handle_data()`, используя указатель `server_pointer`, содержащийся внутри вспомогательного объекта **Thread_Args**, переданного методу `svc()`, как показано на рис. 26.12.

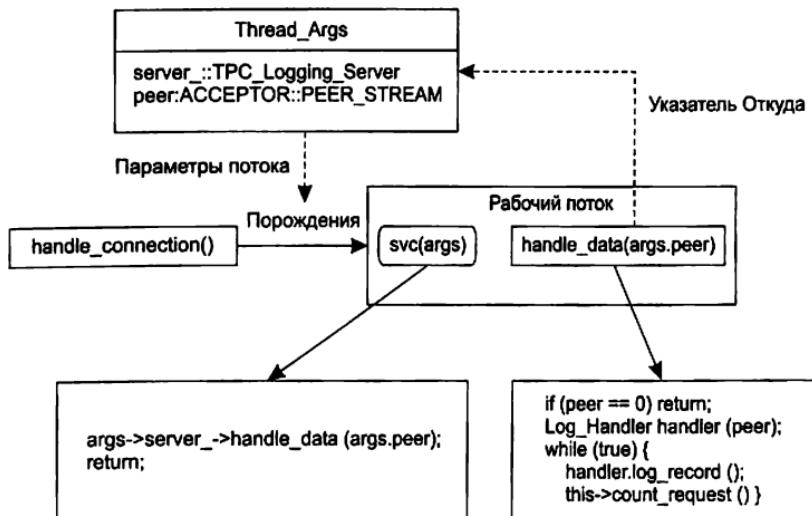


Рис. 26.12. Поведение потока при использовании отдельного потока на каждое подключение

Следующий код показывает реализацию основной программы **TPC_Logging_Server**, которая использует защищенный socket API и блокировку чтения/записи:

```

int main (int argc, char *argv[]) {
    TPC_Logging_Server<SSL_Acceptor, RW_Lock> server (argc, argv);
    server.run ();
}

```

```
    return 0;  
}
```

Функция `main()` подвергает обработке `TPC_Logging_Server`, который соединяется с помощью SSL-подключений и использует `RW_Lock` для синхронизации функции `count_connections()` в базовом классе `Logging_Server`. За исключением имени обрабатываемого класса, эта функция `main()` идентична той, что была написана ранее в этой главе для `Reactive_Logging_Server`. Эти общие свойства являются еще одним аспектом красоты нашей конструкции: независимо от выбранной нами конкретной комбинации параллельной обработки, IPC и механизмов синхронизации, обработка и инициирование работы нашего сервера остаются без изменений.

Служба регистрации, использующая отдельный поток на каждое подключение, предназначена для снятия ограничений расширяемости присущих последовательной реализации, рассмотренных ранее в разделе «Оценка последовательных решений сервера регистрации». Конструкция нашей объектно-ориентированной рабочей среды позволяет сравнительно просто придать этой параллельной модели законченный вид с минимальными изменениями существующего программного кода. В частности, `TPC_Logging_Server` наследует реализацию `open()`, `count_request()` и, что наиболее важно, `run()`, позволяя этому классу получить вполне очевидный выигрыш за счет устранений ошибок и улучшений нашего основного цикла событий. Кроме того, добавление необходимой синхронизации для `request_count_` достигается простой параметризацией `TPC_Logging_Server` классом `RW_LOCK`.

Сервер регистрации, использующий отдельный процесс на каждое подключение

Рассматриваемый далее сервер регистрации, использующий отдельный процесс на каждое подключение, похож на конструкцию, использовавшую отдельный поток на каждое подключение, показанную на рис. 26.10, за исключением того что для обработки входящей регистрационной записи от каждого клиента вместо порождения потока мы порождаем новый процесс. Выбор процессов вместо потоков для параллельной обработки заставляет нас выбирать конструкции, которые можно приспособить к вариантам семантики создания процессов, имеющихся у различных платформ. Процессы в API Linux и Windows имеют два основных семантических различия, которые должны быть инкапсульированы нашим сервером.

- В Linux (и других POSIX-системах) основным методом создания новых процессов является системная функция `fork()`, которая генерирует точный дубликат образа вызывающей программы, включая открытые обработчики ввода–вывода. Процессы отличаются лишь своими возвращаемыми из `fork()` значениями. На данном этапе дочерние процессы могут выбрать продолжение с этого самого места или загрузить другой образ программы, используя семейство системных вызовов `exec*()`.
- А в Windows используется API-вызов `CreateProcess()`, являющийся функциональным эквивалентом POSIX `fork()`, за которым тут же следует вызов

одной из системных функций `exec*`(). В силу этого различия в Windows вы получаете *полностью новый процесс*, который по умолчанию *не имеет* доступа к обработчикам ввода–вывода, открытых в родительском процессе. Поэтому чтобы воспользоваться подключениями, принятыми родительским процессом, обработчики должны быть явным образом продублированы и переданы дочернему процессу по командной строке.

Вследствие этого мы определяем ряд фасадов оболочек, которые не только скрывают синтаксические различия между платформами, но и к тому же предоставляют способ скрытия семантических различий. Эти оболочки состоят из трех взаимодействующих классов, показанных на рис. 26.13. Класс `Process` представляет отдельный процесс и используется для создания и синхронизации процессов. Класс `Process_Options` предоставляет способ установки как независимых от применяемой платформы параметров обработки (таких как параметры командной строки и переменные окружения), так и специфических для платформы параметров (таких как аннулирование зомби-процессов). И наконец, класс `Process_Manager` управляет жизненным циклом групп процессов, используя способ, переносимый с платформы на платформу. В этой главе мы не будем рассматривать все возможности применения этих фасадов оболочек, но следует заметить, что они основаны на фасадах оболочек, имеющихся в ACE¹. Нам достаточно знать, что не только процессы могут создаваться переносимыми на Linux и Windows, но и обработчики ввода–вывода могут быть продублированы и автоматически переданы новому процессу переносимым способом.

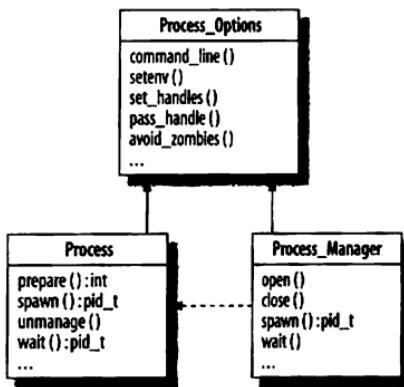


Рис. 26.13. Фасады оболочек переносимого процесса

Поэтому задача проектирования состоит в восприятии факта, что порождение процессов после приема новых подключений начнется в самом начале нашей программы. Мы, конечно, не хотим, чтобы дочерние процессы пытались открыть новый получатель и самостоятельно отслеживать подключения; вместо

¹ Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming*. Vol. 1: Mastering Complexity with ACE and Patterns, Addison-Wesley, 2001.

этого они должны отслеживать наступление событий данных только лишь в связанных с ними обработчиках. В простейшем решении этой задачи для определения этих условий и вызова специальной точки входа, определенной в интерфейсе для нашего основанного на процессах класса `Logging_Server`, можно было бы положиться на приложения.

Разумеется, это простейшее решение далеко от идеала. Оно потребует от нас не только изменить общедоступный интерфейс, базирующийся на процессах `Logging_Server`, но и обнажить внутренние детали реализации приложения, нарушив тем самым инкапсуляцию. Более подходящее решение состоит в замене шаблонного метода `run()`, унаследованного от базового класса `Logging_Server`, который передаст копию параметров командной строки пользователю для определения, были ли им переданы какие-либо обработчики ввода-вывода. Если нет, процесс посчитает себя родительским и передаст полномочия методу `run()` базового класса. В противном случае процесс посчитает себя дочерним, после чего декодирует обработчик и вызовет `handle_data()`, как показано на рис. 26.14.

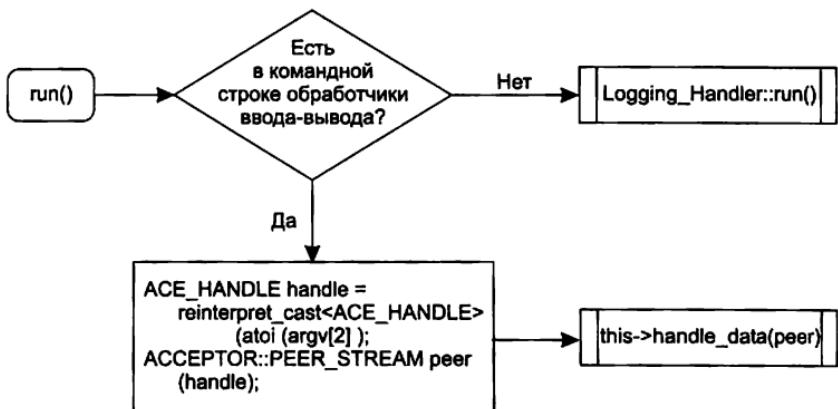


Рис. 26.14. Шаблонный метод `run()` при использовании отдельного процесса на каждое подключение

Оставшаяся часть реализации сервера не должна вызвать особых трудностей. Как показано на рис. 26.15, фасад оболочки процесса делает процедуру рождения нашего рабочего процесса достаточно простой. Реализация `handle_data()` должна быть абсолютно идентична той, что показана на рис. 26.12.

Наша новая реализация метода `run()` из базового класса `Logging_Server` позволяет сохранить красоту, присущую простоте, прямолинейности, и универсальность вызова, используемого нашими другими серверами регистрации:

```

int main (int argc, char *argv[])
{
    PPC_Logging_Server<SSL_Acceptor, Null_Mutex> server (argc, argv);
    server.run();
    return 0;
}
  
```

Эта `main()`-программа отличается от той, что использовалась в сервере, имевшем отдельный поток на каждое подключение, только именем обработы-

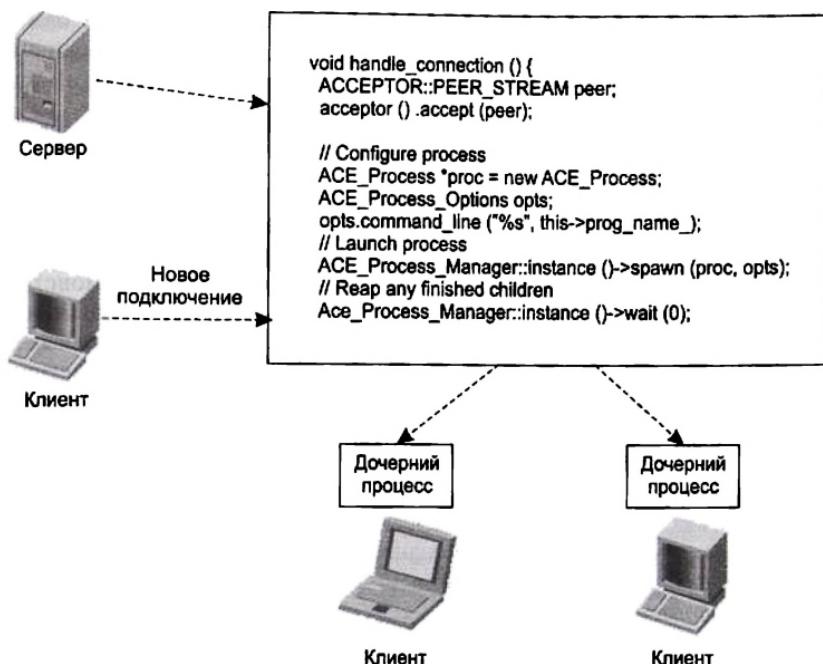


Рис. 26.15. Обработка подключения для сервера, использующего отдельный процесс на каждое подключение

ваемого класса и выбором для синхронизации `Null_Mutex`. Диспетчеризация родительского или дочернего процесса осуществляется в явном виде в методе `run()`, управляемом параметрами командной строки, переданными конструктору `PPC_Logging_Server`.

Оценка параллельных решений сервера регистрации

Рассмотренные в этом разделе два сервера регистрации с параллельной обработкой данных существенно улучшают `Reactive_Logging_Server` и `Iterative_Logging_Server` по возможностям расширения при увеличении количества клиентов и делают это за счет использования аппаратной и программной (со стороны оперативной системы) поддержки выполнения нескольких потоков. Но разработка параллельных стратегий, использующих поток или процесс на каждое подключение и позволяющих не брать в расчет используемую платформу, дается нелегко. Мы выполнили эту задачу за счет использования фасадов оболочек для скрытия различных платформ. Конструкция нашего сервера, основанная на использовании рабочей среды, также обеспечивает общий внешний интерфейс для класса `Logging_Server`, ограждая большую часть сервера регистрации от сконфигурированной стратегии параллельной обработки. Кроме того, наша

конструкция усиливает шаблонный метод `run()`, унаследованный от базового класса `Logging_Server`, позволяя нашим реализациям использовать общие преимущества от устранения ошибок или других улучшений цикла событий основного сервера.

Вывод

Приложение сервера регистрации, представленное в этой главе, дает нам легкое в усвоении, но вполне реалистичное средство, позволяющее продемонстрировать применение объектно-ориентированных технологий проектирования и программирования, паттернов и рабочих сред для создания программ сетевых приложений. В частности, наша объектно-ориентированная среда демонстрирует ряд красивых элементов конструкции, выстроенных от абстрактного проекта к конкретным элементам в примерах реализации различных моделей параллельной обработки. В нашей конструкции также используются такие свойства C++, как шаблоны и виртуальные функции, в связке с такими паттернами проектирования, как Фасад оболочки (`Wrapper Facade`) и Шаблонный метод (`Template Method`), применяемыми для создания семейства серверов регистрации, обладающих свойствами переносимости, многократности использования, гибкости и наращиваемости.

Паттерн Шаблонный метод в методе `run()`, принадлежащем базовому классу `Logging_Server`, позволяет нам определить общие шаги в сервере регистрации, уступая специализацию отдельных шагов его действий методам-ловушкам в производных классах. Хотя этот паттерн помогает «вынести за скобки» в базовый класс общие шаги, его нельзя в полной мере применить ко всем необходимым нам задачам по внесению изменений, в частности в механизмы синхронизации и IPC. Поэтому для этих оставшихся нерешенными аспектов общей задачи мы воспользовались паттерном Фасада оболочки, чтобы скрыть семантические и синтаксические различия, делая эти аспекты в конечном счете полностью независимыми от реализации отдельных моделей параллельной обработки. Эта конструкция позволяет нам использовать параметризированные классы для обращения к этим аспектам изменчивости, повышающим гибкость нашей рабочей среды, не нанося вреда производительности.

И наконец, наши отдельные реализации параллельных моделей, к которым относятся модели с отдельным потоком на каждое подключение и с отдельным процессом на каждое подключение, использовали фасады оболочек для придания их реализациям большей элегантности и переносимости. В конечном счете мы получили архитектуру программного обеспечения, существенно экономящую наши силы, позволяющую разработчикам многократно использовать общие конструкции и образцы программного кода, а также предоставить средства инкапсуляции вариативности общим, параметризуемым способом.

Конкретную реализацию этой рабочей среды сервера регистрации можно найти здесь: http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/ACE/examples/Beautiful_Code и в пакете распространения ACE, в файле `ACE_wrappers/examples/Beautiful_Code`.

27 Объединение деловых партнеров с помощью RESTful

Эндрю Патцер (*Andrew Patzer*)

Несколько лет назад, когда я работал консультантом, был период продолжительностью год или два, когда казалось, что каждый клиент, с которым я беседовал, был абсолютно уверен в необходимости для своего бизнеса какого-нибудь решения, связанного с веб-службами. Разумеется, немногие из моих клиентов на самом деле понимали, что это означает или зачем им может понадобиться подобная архитектура, но поскольку они постоянно слышали о веб-службах Интернета, встречали информацию о них в журналах, на коммерческих выставках, то, по их мнению, лучше было сесть в этот автобус, пока еще не поздно.

Не поймите меня превратно. Я не против веб-служб. Я просто не слишком большой сторонник принимать технические решения, основываясь лишь на стремлении во чтобы то ни стало следовать текущей моде. Эта глава будет посвящена некоторым доводам в пользу применения архитектуры веб-служб, а также исследованию некоторых вариантов, которые стоит рассмотреть при интегрировании своих систем с окружающим миром.

В этой главе мы изучим реально существующий проект, в котором деловому партнеру предоставлен набор служб, и обсудим ряд конструктивных решений, которые были приняты в процессе разработки этого проекта. Задействованные технологии включают Java (J2EE), XML, протокол электронного бизнеса Rosettanet и библиотеку функций, используемую для связи с программами, запущенными на системе AS/400. При демонстрации способов придания системе расширяемости для будущих распространителей, которые могут использовать другие протоколы и нуждаться в доступе к другим службам, я также рассмотрю вопрос использования интерфейсов и шаблонов проектирования типа фабрики – factory design pattern.

Предыстория проекта

Проект, о котором пойдет речь в этой главе, был инициирован звонком одного из наших клиентов: «Нам нужен набор веб-служб, чтобы объединить наши

системы с одним из наших распространителей». Клиентом был крупный производитель электротехнических компонентов. Система, на которую он ссылался, была MAPICS – производственная система, написанная на языке RPG и работающая на его машинах AS/400. Основной распространитель обновил свою собственную бизнес-систему и нуждался в модификации способов связи с системой управления заказами для получения сведений о наличии продукта и состоянии выполнения заказа.

В прежние времена оператор распространителя осуществлял простое удаленное подключение к системе AS/400 производителя и для доступа к нужным экранам нажимал «клавишу быстрого вызова» (насколько я помню, F13 или F14). Чуть позже в программном коде вы увидите, что разработанную для них новую систему назвали *hotkey* («горячая клавиша»), и с тех пор это вошло в их лексикон, примерно так же, как слово *google* стало современным глаголом.

Теперь, когда распространитель ввел в действие новую систему электронного бизнеса, понадобился автоматизированный способ объединения данных производителя с системой распространителя. Поскольку для моего клиента это был всего лишь один из распространителей, хотя и самый крупный, система должна была позволять в будущем добавлять и других распространителей, независимо от имеющихся у них протоколов и запросов. Другим определяющим фактором был относительно низкий уровень профессиональных навыков персонала, предназначавшегося для обслуживания и расширения этого программного обеспечения. Несмотря на их неплохую квалификацию в других областях, разработка на Java (равно как и другие разновидности веб-разработки) для них все еще была в новинку. Итак, я понимал, что мое творение должно быть простым и легко расширяемым продуктом.

Предоставление служб внешним клиентам

До работы над этим проектом я провел ряд технических презентаций для групп пользователей и несколько конференций, посвященных протоколу обмена структурированными сообщениями в распределенной вычислительной среде – SOAP (Simple Object Access Protocol – простой протокол доступа к объектам) и архитектуре веб-служб. Поэтому когда мне позвонили, казалось, что я как нельзя лучше подхожу для выполнения поставленной клиентом задачи. Как только я понял, что им на самом деле нужно, я решил, что было бы намного лучше начать с набора служб, предоставляемых через простые GET- и POST-запросы по протоколу HTTP, обмениваясь XML-данными с описаниями запросов и ответов. Хотя к тому времени я еще не знал, что это архитектурный стиль, который теперь общеизвестен как REST, или передача репрезентативного состояния – *Representational State Transfer*.

Как я пришел к решению использовать REST через SOAP? Для принятия решения о выборе архитектуры веб-служб нужно рассмотреть следующий перечень вопросов.

Сколько различных систем потребуют доступ к этим службам и все ли они известны на данный момент?

Этому производителю было известно лишь об одном распространителе, нуждающемся в доступе к его системам, но он также знал и о том, что и другие распространители могут в будущем решиться на то же самое.

Есть ли у вас ограниченный круг конечных пользователей, заранее осведомленных об этих службах, или же эти службы должны предоставлять свое описание неизвестным пользователям для автоматического подключения?

Поскольку между производителем и всеми распространителями его продукции должны быть вполне определенные отношения, была гарантия, что любой потенциальный пользователь будет заранее знать, как получить доступ к системам производителя.

Какой тип структуры должен поддерживаться во время отдельной транзакции? Должен ли запрос зависеть от результатов предыдущего запроса?

В нашем случае каждая транзакция будет состоять из отдельного запроса и соответствующего ему результата, который не зависит ни от чего другого.

Ответы на предыдущие вопросы относительно данного проекта привели к очевидному выбору простого предоставления известных служб посредством использования HTTP-протокола и расширения данных с использованием стандартного протокола электронного бизнеса, понятного обеим системам. Если бы производитель предпочел разрешить неизвестным пользователям запрашивать наличие изделий, то я мог бы выбрать полноценное SOAP-решение, потому что это позволило бы системам обнаруживать службы и программно связываться с ними без предварительного знания системы.

Сейчас я работаю в области биоинформатики, где есть определенная потребность в SOAP-стиле архитектуры веб-служб. Мы используем проект под названием BioMoby (<http://www.biomoby.org>) для определения веб-служб и их публикации в центральном хранилище, позволяющем другим группам буквально перетаскивать наши службы в технологический процесс, создающий конвейеры данных, помогающие биологам объединять различные наборы данных и проводить различную аналитическую работу над полученными результатами. Это хороший аргумент в пользу выбора SOAP через REST. Неизвестные пользователи могут обращаться к вашим данным и инструментам, заранее даже не зная об их существовании.

Определение интерфейса службы

Первое, что я сделал, когда принял решение о том, как реализовать это программное обеспечение, — это определил, как пользователь будут составлять запросы и получать ответы. После разговора с техническими представителями распространителя (основного пользователя) я понял, что их новая система способна посыпать XML-документы через HTTP POST-запрос и анализировать результаты, полученные в виде XML-документа. XML должен был быть пред-

ставлен в формате, соответствующем протоколу электронного бизнеса Rosettanet (к чему мы еще вернемся), но на тот момент вполне достаточно было знать, что их система способна держать связь по протоколу HTTP путем обмена запросами и ответами в формате XML. На рис. 27.1 показана общая схема взаимодействия между каждой из систем.

В недавнем прошлом производитель был приобретен более крупной корпорацией, которая предписала использование продукции компании IBM во всей своей организации. Поэтому я уже знал, какой сервер приложения и соответствующую ему технологию нужно использовать. Я реализовал интерфейс службы в виде Java-сервлета, работающего на сервере приложений IBM WebSphere. Знание того, какое программное обеспечение понадобится для доступа к функциям, запускаемым на сервере AS/400 при использовании API, основанном на языке Java, облегчило мне принятие такого решения.

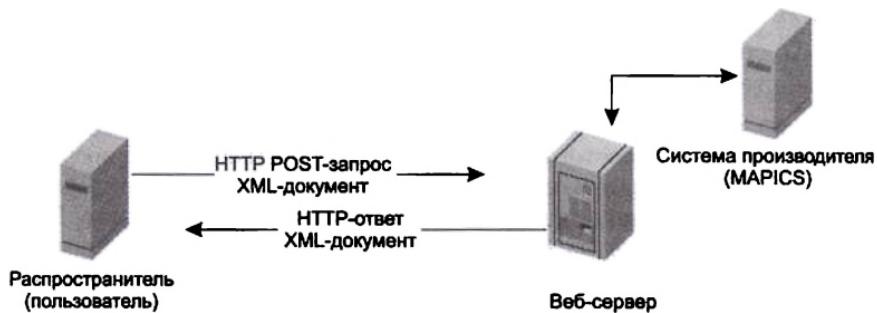


Рис. 27.1. Интерфейс службы и внутренних систем

В файле web.xml, описывающем сервлет, который обеспечит необходимый пользователям интерфейс, находится следующий код:

```

<servlet>
    <servlet-name>HotKeyService</servlet-name>
    <display-name>HotKeyService</display-name>
    <servlet-class>comxxxxxxxxxx.hotkey.Service</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HotKeyService</servlet-name>
    <url-pattern>/HotKeyService</url-pattern>
</servlet-mapping>
    
```

Сам сервлет управляет только POST-запросами, подменяя метод doPost интерфейса Servlet и обеспечивая реализацию по умолчанию стандартных методов жизненного цикла. В следующем примере кода показана полная реализация службы, но обычно, когда я впервые приступаю к анализу проблемы и построению решения, я пишу в коде серию комментариев в качестве мест, куда позже будет помещен настоящий код. Затем я систематически атакую каждый псевдо-кодовый комментарий, пока не получу работоспособную реализацию. Это помогает мне сосредоточиться на том, как каждая часть программы относится ко всему решению в целом:

```
public class Service extends HttpServlet implements Servlet {  
  
    public void doPost(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
  
        // Считывание данных запроса и сохранение их в StringBuffer  
        BufferedReader in = req.getReader();  
        StringBuffer sb = new StringBuffer();  
        String line;  
        while ((line = in.readLine())!= null) {  
            sb.append(line);  
        }  
  
        HotkeyAdaptor hotkey = null;  
  
        if (sb.toString().indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {  
            // Запрос цены и наличия  
            hotkey = HotkeyAdaptorFactory.getAdaptor(  
                HotkeyAdaptorFactory.ROSETTANET,  
                HotkeyAdaptorFactory.PRODUCTAVAILABILITY);  
        }  
        else if (sb.toString().indexOf("Pip3A5PurchaseOrderStatusQuery ") > 0) {  
            // Состояние заказа  
            hotkey = HotkeyAdaptorFactory.getAdaptor(  
                HotkeyAdaptorFactory.ROSETTANET,  
                HotkeyAdaptorFactory.ORDERSTATUS);  
        }  
  
        boolean success = false;  
  
        if (hotkey != null) {  
            /* Передача данных XML-запроса */  
            hotkey.setXML(sb.toString());  
            /* Анализ данных запроса */  
            if (hotkey.parseXML()) {  
                /* Выполнение программы AS/400 */  
                if (hotkey.executeQuery()) {  
                    /* Возврат XML-ответа */  
                    resp.setContentType("text/xml");  
                    PrintWriter out = resp.getWriter();  
                    out.println(hotkey.getResponseXML());  
                    out.close();  
                    success = true;  
                }  
            }  
        }  
  
        if (!success) {  
            resp.setContentType("text/xml");  
            PrintWriter out = resp.getWriter();  
            out.println("Error retrieving product availability.");  
            out.close();  
        }  
    }  
}
```

Изучая этот код, можно заметить, что он сначала считывает данные запроса и сохраняет их для дальнейшего использования. Затем он исследует эти данные, чтобы определить, к какому типу запроса они относятся: цены и наличия или выяснения состояния заказа. После определения типа запроса создается соответствующий вспомогательный объект. Обратите внимание, как я воспользовался интерфейсом HotkeyAdaptor, чтобы предусмотреть сразу несколько реализаций и исключить необходимость создания дублированных фрагментов кода для каждого типа запроса.

Остальная часть кода этого метода занимается синтаксическим анализом данных XML-запроса, выполнением соответствующего запроса на системе AS/400, созданием XML-ответа и отправкой его пользователю через HTTP-протокол. В следующем разделе вы увидите, как я скрыл детали реализации, используя интерфейсы и весьма широко распространенный шаблон проектирования типа Factory (фабрика).

Маршрутизация службы с использованием шаблона Factory

Одним из требований, предъявлявшихся к этой системе, была возможность подстраивания под широкий спектр будущих запросов от нескольких различных типов систем с минимальными затратами на программирование. Я полагаю, что выполнил это требование за счет сведения реализации к одной-единственной команде интерфейса, которая обнаруживает основные методы, необходимые для ответа на широкий круг запросов:

```
public interface HotkeyAdaptor {  
  
    public void setXML(String _xml);  
    public boolean parseXML();  
    public boolean executeQuery();  
    public String getResponseXML();  
}
```

Каким же образом сервлет решает, какую конкретно реализацию интерфейса подвергать обработке? Сначала он смотрит в данные запроса и ищет в них определенную строку, которая сообщит ему, какой это тип запроса. Затем он использует статический метод объекта-фабрики (*factory object*), чтобы выбрать соответствующую реализацию.

Поскольку сервлет это знает, то независимо от используемой нами реализации, каждому из этих методов будут предоставлены соответствующие ответы. Благодаря использованию интерфейса в основном сервлете нам остается лишь единожды написать исполняемый код, для которого все равно, с каким типом запроса он имеет дело или кто мог сделать этот запрос. Все детали скрыты в каждой конкретной реализации этого интерфейса. Вот еще один фрагмент кода, относящийся к сервлету:

```
HotkeyAdaptor hotkey = null;  
  
if (sb.toString().indexOf("Pip3A2PriceAndAvailabilityRequest") > 0) {
```

```

// Запрос цены и наличия
hotkey = HotkeyAdaptorFactory.getAdaptor(
    HotkeyAdaptorFactory.ROSETTANET,
    HotkeyAdaptorFactory.PRODUCTAVAILABILITY);
}

else if (sb.toString().indexOf("Pip3A5PurchaseOrderStatusQuery ") > 0) {
    // Состояние заказа
    hotkey = HotkeyAdaptorFactory.getAdaptor(
        HotkeyAdaptorFactory.ROSETTANET,
        HotkeyAdaptorFactory.ORDERSTATUS);
}

```

Объект-фабрика HotkeyAdaptorFactory содержит статический метод, воспринимающий два параметра, сообщающих ему, какой XML-протокол использовать и какой это тип запроса. Это определено в виде статических констант в самом объекте-фабрике. Как показано в следующем коде, объект-фабрика просто использует оператор switch, чтобы выбрать соответствующую реализацию:

```

public class HotkeyAdaptorFactory {

    public static final int ROSETTANET = 0;
    public static final int BIZTALK = 1;
    public static final int EBXML = 2;

    public static final int PRODUCTAVAILABILITY = 0;
    public static final int ORDERSTATUS = 1;

    public static HotkeyAdaptor getAdaptor(int _vocab, int _target) {

        switch (_vocab) {
            case (ROSETTANET) :
                switch (_target) {
                    case (PRODUCTAVAILABILITY) :
                        return new HotkeyAdaptorRosProdAvailImpl();
                    case (ORDERSTATUS) :
                        return new HotkeyAdaptorRosOrdStatImpl();
                    default :
                        return null;
                }
            case (BIZTALK) :
            case (EBXML) :
                default :
                    return null;
        }
    }
}

```

Хотя это может показаться довольно простой абстракцией, но она играет весьма существенную роль, позволяя не слишком опытному персоналу программистов легко читать код и понимать его предназначение. Когда настанет время добавить нового распространителя, который, скажем, привык к использованию продукта BizTalk компании Microsoft и хочет размещать заказы электронным способом, у программиста уже есть простой шаблон, чтобы добавить к системе новое требование.

Обмен данными с использованием протоколов электронного бизнеса

Одной из новинок в этом проекте для меня стало использование стандартных протоколов электронного бизнеса. Когда распространитель сообщил мне о требовании по обмену запросами и ответами с использованием стандарта Rosettanet, мне пришлось провести небольшое исследование. Сначала я посетил веб-сайт Rosettanet (<http://www.rosettanet.org>) и загрузил те стандарты, которые меня интересовали. Так я нашел диаграмму, детализирующую типовой обмен между деловыми партнерами, а также технические требования для запросов и ответов в формате XML.

Поскольку мне пришлось проделать большой объем работы, действуя методом проб и ошибок, первое, с чего я начал, — это создание теста, который я сам мог запустить для имитации взаимодействия с распространителем, не требующий согласования тестирования с его персоналом на каждом этапе разработки. Для управления HTTP-обменом я воспользовался системой Apache Commons HttpClient:

```
public class TestHotKeyService {  
  
    public static void main (String[] args) throws Exception {  
  
        String strURL = "http://xxxxxxxxxx/HotKey/HotKeyService":  
        String strXMLFilename = "SampleXMLRequest.xml":  
        File input = new File(strXMLFilename);  
  
        PostMethod post = new PostMethod(strURL):  
        post.setRequestBody(new FileInputStream(input));  
        if (input.length() < Integer.MAX_VALUE) {  
            post.setRequestContentLength((int)input.length()):  
        } else {  
            post.setRequestContentLength(  
                EntityEnclosingMethod.CONTENT_LENGTH_CHUNKED):  
        }  
  
        post.setRequestHeader("Content-type", "text/xml; charset=ISO-8859-1");  
  
        HttpClient httpclient = new HttpClient():  
        System.out.println("[Response status code]: " +  
            httpclient.executeMethod(post));  
        System.out.println("\n[Response body]: "):  
        System.out.println("\n" + post.getResponseBodyAsString());  
  
        post.releaseConnection():  
    }  
}
```

Это позволило мне ускорить свое обучение, поскольку я подверг испытаниям несколько различных типов запросов и изучил полученные результаты. Я твердый сторонник самого раннего погружения в программирование. По книгам, публикациям на веб-сайтах или подборке API-документации вы можете

всего лишь чему-нибудь научиться. Но приложив свои руки к самому процессу разработки, вы откроете для себя массу нового, о чём даже не задумывались, изучая проблему.

Стандарт Rosettanet, как и многие другие стандарты, крайне детализирован и проработан. Возможно, для решения какой-нибудь конкретной задачи вам понадобится лишь его малая часть. Для этого проекта мне пригодился только набор из нескольких стандартных идентификационных файлов, а также номер изделия и дата для запросов цены или номер заказа для запросов о его состоянии.

Анализ XML с использованием XPath

Данные XML-запроса сильно отличались от простого XML. Я уже упоминал, что стандарт Rosettanet весьма досконально проработан. Если бы не было XPath, то анализ такого документа превратился бы в сущий кошмар. Используя XPath-отображение, я мог определить точный маршрут к каждому интересующему меня узлу и без особого труда извлечь нужные данные. Я выбрал реализацию этого отображения в виде HashMap, путем последующего перебора которого использовал определенные узлы и создавал новый HashMap, содержащий значения. Затем эти значения использовались в обоих методах – executeQuery и getResponseXML, которые будут рассмотрены позже:

```
public class HotkeyAdaptorRosProdAvailImpl implements HotkeyAdaptor {  
  
    String inputFile; // XML запроса  
    HashMap requestValues; // сохранение разобранных XML-значений из запроса  
    HashMap as400response; // сохранение возвращенного параметра из вызова RPG  
  
    /* Объявление Xpath-отображения и заполнение его статическим блоком  
       инициализации */  
    public static HashMap xpathMappings = new HashMap();  
    static {  
        xpathMappings.put("from_ContactName",  
"//Pip3A2PriceAndAvailabilityRequest/fromRole/PartnerRoleDescription/  
ContactInformation/contactName/FreeFormText");  
        xpathMappings.put("from_EmailAddress", "//Pip3A2PriceAndAvailabilityRequest/  
fromRole/PartnerRoleDescription/ContactInformation/EmailAddress");  
    }  
    // Остальная часть xpath-отображения для краткости опущена ...  
  
    public HotkeyAdaptorRosProdAvailImpl() {  
        this.requestValues = new HashMap();  
        this.as400response = new HashMap();  
    }  
  
    public void setXML(String _xml) {  
        this.inputFile = _xml;  
    }  
  
    public boolean parseXML() {  
  
        try {  
            Document doc = null;
```

```

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
StringReader r = new StringReader(this.inputFile);
org.xml.sax.InputSource is = new org.xml.sax.InputSource(r);
doc = db.parse(is);

Element root = doc.getDocumentElement();

Node node = null;

Iterator xpathvals = xpathmappings.values().iterator();
Iterator xpathvars = xpathmappings.keySet().iterator();
while (xpathvals.hasNext() && xpathvars.hasNext()) {
    node = XPathAPI.selectSingleNode(root, String.xpathvals.next());
    requestValues.put((String)xpathvars.next(),
        node.getChildNodes().item(0).getNodeValue());
}
}

catch (Exception e) {
    System.out.println(e.toString());
}

return true;
}

public boolean executeQuery() {
    // Код опущен...
}

public String getResponseXML() {
    // Код опущен...
}
}

```

Метод executeQuery содержит весь код, необходимый для доступа к RPG-коду, запущенному на системе AS/400, в целях получения данных ответа, которые затем будут использованы для построения ответного XML-документа. Много лет назад я работал над проектом, который объединял MAPICS-систему (RPG на AS/400) с новой системой, которую я написал на Visual Basic. Я написал код для обеих обменивающихся сторон, на RPG и CL для AS/400 и на Visual Basic для PC. Это привело к нескольким беседам, в ходе которых я попытался показать множеству RPG-программистов, как объединить их прежние системы с современным клиент-серверным программным обеспечением. В те времена это действительно была сложная и почти что мистическая задача.

С тех пор IBM существенно облегчила ее решение и предоставила библиотеку Java-функций, которые все делают за нас. (Она одна могла бы заменить для меня все многочисленные консультации и книги!) Вот как выглядит код, который использует эту значительно улучшенную библиотеку от IBM:

```

public boolean executeQuery() {

    StringBuffer sb = new StringBuffer();

    sb.append(requestValues.get("from_ContactName")).append("|");

```

```
sb.append(requestValues.get("from_EmailAddress")).append("|");
sb.append(requestValues.get("from_TelephoneNumber")).append("|");
sb.append(requestValues.get("from_BusinessIdentifier")).append("|");
sb.append(requestValues.get("prod_BeginAvailDate")).append("|");
sb.append(requestValues.get("prod_EndAvailDate")).append("|");
sb.append(requestValues.get("prod_Quantity")).append("|");
sb.append(requestValues.get("prod_ProductIdentifier")).append("|");

try {
    AS400 sys = new AS400("SS100044", "ACME", "MOUSE123");

    CharConverter ch = new CharConverter();
    byte[] as = ch.stringToByteArray(sb.toString());

    ProgramParameter[] parmList = new ProgramParameter[2];
    parmList[0] = new ProgramParameter(as);
    parmList[1] = new ProgramParameter(255);

    ProgramCall pgm = new ProgramCall(sys,
        "/QSYS.LIB/DEVOBJ.LIB/J551231.PGM", parmList);
    if (pgm.run() != true) {
        AS400Message[] msgList = pgm.getMessageList();
        for (int i=0; i < msgList.length; i++) {
            System.out.println(msgList[i].getID() + " : " +
                msgList[i].getText());
        }
    } else {
        CharConverter chconv = new CharConverter();
        String response =
            chconv.byteArrayToString(parmList[1].getOutputData());

        StringTokenizer st = new StringTokenizer(response, "|");

        String status = (String) st.nextToken().trim();
        as400response.put("Status", status);
        String error = (String) st.nextToken().trim();
        as400response.put("ErrorCode", error);
        String quantity = (String) st.nextToken().trim();
        as400response.put("Quantity",
            String.valueOf(Integer.parseInt(quantity)));

        if (status.toUpperCase().equals("ER")) {
            if (error.equals("1")) {
                as400response.put("ErrorMsg",
                    "Account not authorized for item availability.");
            }
            if (error.equals("2")) {
                as400response.put("ErrorMsg", "Item not found.");
            }
            if (error.equals("3")) {
                as400response.put("ErrorMsg", "Item is obsolete.");
                as400response.put("Replacement",
                    (String) st.nextToken().trim());
            }
        }
    }
}
```

```

        if (error.equals("4")) {
            as400response.put("ErrorMsg",
                "Invalid quantity amount.");
        }
        if (error.equals("5")) {
            as400response.put("ErrorMsg",
                "Preference profile processing error.");
        }
        if (error.equals("6")) {
            as400response.put("ErrorMsg",
                "ATP processing error.");
        }
    }
}
catch (Exception e) {
    System.out.println(e.toString());
}
return true;
}

```

Этот метод начинает свою работу со сборки строки параметров (с вертикальной чертой в качестве разделителя), которая передается в программу AS/400, где проводится анализ строки, извлекаются запрошенные данные и возвращается строка со знаком вертикальной черты в качестве разделителя, в которой содержится состояние и код ошибки, а также результат выполнения операции.

Предполагая, что ошибки не произошло, результат этого взаимодействия с AS/400 сохраняется в другом `HashMap`, которым мы воспользуемся при конструировании ответного документа в формате XML. Если все же произошла ошибка, то вместо этого в ответ будет записано соответствующее сообщение.

Сборка XML-ответа

Я всегда получаю удовольствие, когда вижу, каким большим числом способов люди пытаются программно создать XML-документы. Я всегда утверждал, что XML-документы – это всего лишь большие текстовые строки. Поэтому обычно намного легче просто написать такую строку, используя `StringBuffer`, чем пытаться построить объектную модель документа – DOM (Document Object Model) или использовать специальную библиотеку для генерации XML.

Для этого проекта я просто создал объект `StringBuffer` и добавлял каждую отдельную строку XML-документа, следуя стандарту Rosettanet. В следующем примере кода я опустил несколько строк, но он может подсказать вам идею построения ответа:

```

public String getResponseXML() {

    StringBuffer response = new StringBuffer();
    response.append("<Pip3A2PriceAndAvailabilityResponse>").append("\n");
    response.append(" <ProductAvailability>").append("\n");
    response.append(" <ProductQuantity>").append(as400response.get("Quantity"));
}

```

```
append("</ProductQuantity>").append("\n");
response.append(" </ProductAvailability>").append("\n");
response.append(" <ProductIdentification>").append("\n");
response.append(" <PartnerProductIdentification>").append("\n");
response.append(" <GlobalPartnerClassificationCode>Manufacturer</
GlobalPartnerClassificationCode>").append("\n");
response.append(" <ProprietaryProductIdentifier>").append(requestValues.
get("prod_ProductIdentifier")).append("</ProprietaryProductIdentifier>")
.append("\n");
response.append(" </PartnerProductIdentification>").append("\n");
response.append(" </ProductIdentification>").append("\n");
response.append(" </ProductPriceAndAvailabilityLineItem>").append("\n");
response.append("</Pip3A2PriceAndAvailabilityResponse>").append("\n");

return response.toString();
}
```

Вывод

Просматривая снова этот код, написанный мною свыше двух лет назад, я считаю, что он вполне созрел для пересмотра и продумывания более подходящих способов его написания. Хотя некоторые фрагменты реализации кода можно было бы переписать заново, я думаю, что общая конструкция этого решения была бы такой же. Этот код выдержал испытание временем, поскольку клиент за это время самостоятельно добавил несколько новых распространителей и новые типы запросов, востребовав минимум помощи у посторонних поставщиков услуг, вроде меня.

В настоящее время, являясь начальником отдела биоинформатики, я использовал этот код для демонстрации ряда приемов своему персоналу, когда проводил обучение принципам объектно-ориентированного проектирования и технологии XML-парсинга. Я мог бы написать о коде более поздней разработки, но подумал, что с помощью именно этого кода можно продемонстрировать ряд основных принципов, которые очень важно усвоить начинающим разработчикам программного обеспечения.

28 Красивая отладка

Андреас Целлер (*Andreas Zeller*)

«Меня зовут Андреас, и я занимался отладкой». Добро пожаловать в общество анонимных отладчиков, где вы можете рассказать о своей истории отладки и утешиться, выслушав истории друзей по несчастью. Итак, вы провели еще одну ночь вне дома? Хорошо хоть при этом вы воевали на фронтах отладки. Вы пока не можете сказать своему начальнику, когда будет устранен дефект? Да-вайте надеяться на лучшее! Парень из соседней кабинки хвастается о том, что искал ошибку в течение 36 часов подряд? Впечатляет!

Но в отладке нет совершенно ничего привлекательного. Это гадкий утенок нашей профессии, признание того, что мы далеки от совершенства, наименее предсказуемая или поддающаяся объяснению работа, и постоянный источник чувства вины и досады: «Если бы мы постарались с самого начала, то не пришлось бы застрять в этом хаосе». Упущение — это преступление, а отладка — наказание.

Но предположим, что приложены все усилия, чтобы не допустить ошибок. И все же временами нам придется попадать в ситуации, где без отладки не обойтись. Как и в любом другом деле, нам следует заниматься отладкой профессионально, возможно, даже «красиво».

Но разве может быть в отладке какая-то красота? Я уверен, что может. В моей программистской практике был ряд моментов, когда я видел красоту в отладке. Эти случаи не только помогали мне решить текущую проблему, но и приводили к новым подходам в отладке в целом. Эти подходы не только по-своему красивы, но и способны поднять производительность отладки. Причина в их *системности* — они гарантированно приводят к решению проблемы — и частично даже в *автоматизированности* — они сами делают все необходимое, пока вы занимаетесь решением других задач.

Интересно? Читайте дальше.

Отладка отладчика

Первую встречу с красотой в отладке мне подарила одна из моих студенток. Работая в 1994 году над диссертацией на звание магистра, Доротея Люткехаус (Dorothea Lutkehaus) создала визуальный отладочный интерфейс, предоставляемый визуализацию структуры данных. На рис. 28.1 показана копия экрана созданного ею инструментального средства, названного отладчиком, отображающим данные, — *data display debugger*, или для краткости *ddd*. Когда Доротея показала свой отладчик, все присутствующие, включая меня, были просто поражены: он позволял воспринимать сложные данные буквально за секунды, исследовать их и управлять ими с помощью одной лишь мыши.

Отладчик *ddd* служил оболочкой для использовавшегося в то время отладчика командной строки (имеется в виду GNU-отладчик *gdb*), который был очень мощным, но сложным в использовании инструментом. Поскольку графические пользовательские интерфейсы для разработки программного обеспечения были еще редкостью, *ddd* был своеобразной маленькой революцией. В течение нескольких следующих месяцев мы с Доротеей приложили все усилия, чтобы сделать *ddd* самым красивым отладочным интерфейсом, и в конечном итоге он стал частью экосистемы GNU.

Хотя вести отладку с помощью *ddd* было намного приятнее, чем с инструментом, использующим командную строку, работа с ним не гарантировала превращения в более успешного специалиста по отладке. В этом деле намного важнее не инструмент, а *процесс отладки*. Кстати, *ddd* тоже помог мне в этом убедиться. Все началось с отчета об ошибке *ddd*, полученного мной 31 июля 1998 года и давшего старт второй встрече с красотой отладки. Вот этот отчет:

При использовании DDD с GDB 4.16 команда запуска (*run*) правильно использовала любые предыдущие аргументы командной строки или значения последовательности аргументов. Но когда я перешел к работе с GDB 4.17, эта команда перестала работать: если я вводил команду *run* в окне консоли, предыдущие параметры командной строки утрачивались.

Разработчики *gdb* опять оплошили — выпустили новую версию *gdb*, которая вела себя несколько иначе по сравнению с предыдущей версией. Поскольку *ddd* был внешним интерфейсом, он отсылал команды отладчику *gdb* так же, как это делал бы человек, и анализировал ответы *gdb*, чтобы представить информацию в пользовательском интерфейсе. По-видимому где-то в этом процессе произошел сбой.

Мне пришлось достать и установить новую версию отладчика *gdb*, запустить в качестве его внешнего интерфейса *ddd* и посмотреть, смогу ли я воспроизвести возникновение этой проблемы. Если получится, то нужно будет запустить другой экземпляр отладчика, чтобы вникнуть в суть проблемы. В общем, вполне привычная работа. Но так получилось, что я был уже сыт по горло запущенными отладчиками, отладкой нашего собственного отладчика и в частности тем, что приходилось заниматься отладкой из-за внесенных кем-то изменений.

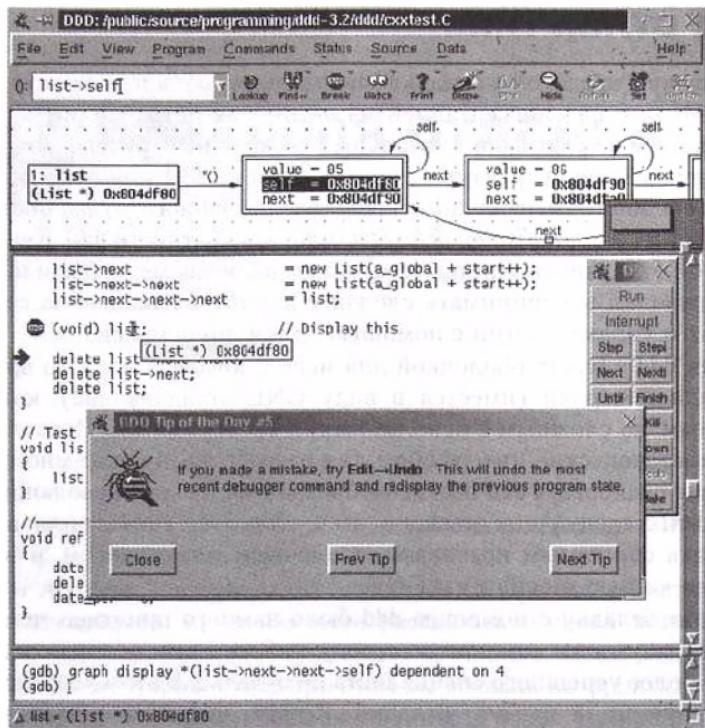


Рис. 28.1. Отладчик ddd в действии

Поэтому я сел и задумался: а нельзя ли решить эту проблему вообще без запуска отладчика? Или можно ли что-нибудь отладить, вообще не занимаясь отладкой?

Поскольку проблема была вызвана изменениями в исходном коде `gdb`, я мог просто посмотреть на код `gdb` или, точнее, на то, чем отличаются две версии отладчика. Я полагал, что изменения в коде напрямую укажут мне на поправки, вызвавшие отказ. Нужно было лишь пропустить оба программных кода через `diff`, средство обнаружения различий. Я так и сделал.

Работа `diff` привела к удивительным результатам. Журнал состоял из 178 200 строк — громадный объем, особенно если учесть, что весь объем исходного кода `gdb` составлял примерно 600 000 строк кода. Не менее чем в 8721 месте исходного кода разработчики что-нибудь вставили, удалили или изменили. Слишком много для выпуска с «несущественными изменениями», что, разумеется, значительно превышало мои возможности. Даже если на проверку отдельного изменения у меня ушло бы 10 секунд, то на поиск проблемных изменений мне пришлось бы затратить 24 часа. Я печально вздохнул, вызвал отладчик и принялся за новый скучный сеанс отладки. Но мысль о более приемлемом пути решения проблемы, о более «красивом» способе отладки, меня не покидала.

Системный подход к процессу отладки

Когда программисты ведут отладку программы, они ищут причину ошибки, которая может крыться в коде, во входных данных или в среде окружения. Эту причину нужно найти и устраниить. Как только она будет устранена, программа заработает. (Если программа откажет и после устранения причины, нам, видимо, придется пересмотреть свой взгляд на нее.)

Основной процесс поиска причин называется *научным методом*. Применительно к отказам программы он работает следующим образом.

1. Ведется наблюдение за отказом программы.
2. Выдвигается *гипотеза* относительно причины отказа, которая согласуется с результатами наблюдения.
3. На основе гипотезы составляются *предсказания*.
4. Предсказания подвергаются тестированию путем проведения экспериментов и дальнейшего наблюдения:
 - если эксперимент и наблюдение удовлетворяют предсказанию, гипотеза уточняется;
 - если нет, ведется поиск альтернативной гипотезы.
5. Шаги 3 и 4 повторяются до тех пор, пока возможности по уточнению гипотезы не будут исчерпаны.

Таким образом, в конечном итоге гипотеза превратится в *теорию*. Это значит, что у вас есть концептуальная структура, объясняющая (и предсказывающая) некоторые аспекты строения вселенной. Ваша отказавшая программа может быть весьма незначительной частичкой вселенной, но все же выработанная теория должна точно предсказать, где именно нужно внести в нее исправления.

Чтобы получить такую теорию, программисты применяют научный метод, поскольку они идут в обратном направлении по причинно-следственной цепочке, которая приводит их к отказу. При этом они:

1. Наблюдают отказ («На выходе возникают неверные данные»).
2. Выдвигают гипотезу относительно причины отказа («Проблема может заключаться в неправильном значении u »).
3. Составляют предсказание («Если u имеет неверное значение, то оно может быть получено из функции $f()$ в строке 632»).
4. Подвергают предсказание тестированию («Действительно, u имеет неверное значение в строке 632»).
5. Делают соответствующие выводы («Значит, $f()$ возвращает неверное значение. Теперь нужно выяснить, откуда оно берется?»).

Среди всевозможных методов, подсказок и уловок последовательное и упорядоченное применение научного метода является ключом к становлению мастерства отладки. Это означает, что нужно соблюдать три важных момента.

Быть точным

Сформулируйте свою гипотезу как можно точнее. Запишите или объясните суть возникшей проблемы другим людям. Ведите записи своих гипотез и наблюдений, чтобы можно было прервать работу и вернуться к ней утром на свежую голову.

Придерживаться системного подхода

Отдавайте отчет своим действиям. Не исследуйте (или не изменяйте) что-нибудь наобум, не имея стройной гипотезы и выведенного из нее предсказания. Убедитесь в том, что возможная причина отказа вами не упущена.

Сначала рассматривать наиболее вероятные причины

Научный метод гарантирует обнаружение причины, но он не сообщает, когда это произойдет. Идентифицируйте сначала наиболее вероятные причины отказа, а затем сконцентрируйтесь на тех из них, которые имеют наибольшие предпосылки к успеху и требуют наименьших усилий на устранение.

К сожалению, отладчики, работающие в диалоговом режиме, по своей сути не поддерживают научный метод. Разумеется, отладчики являются превосходными инструментальными средствами для произвольного просмотра и исследования кода, а также результатов его работы. Они, конечно, хороши, но только для опытных программистов, которые знают, как использовать отладчики по определенной методике. Я предпочел бы видеть программистов, обученных методичным способам отладки, а не умеющих пользоваться какими-нибудь вычурными отладочными средствами. (И я по-прежнему испытываю чувство вины за то, что собственноручно создавал подобные средства.)

Проблемы поиска

Вернемся к первоначальной задаче отладки отладчика. Даже после обнаружения места дефекта и его устранения мне хотелось знать: нет ли какого-нибудь способа автоматического обнаружения дефектных изменений? Нужен был тест, автоматически запускающийся при внесении любых изменений. Как только он не будет пройден, мы будем знать, что изменилось в последнюю очередь, и сможем сразу же устраниТЬ дефект. (Несколько лет спустя Дэвид Сафф (David Saff) и Майкл Эрнст (Michael Ernst) реализовали эту идею под именем *непрерывного тестирования*.)

В моем случае я знал о том изменении, которое вызвало отказ теста, — это было изменение версии с `gdb 4.16` на `gdb 4.17`. Проблема состояла в том, что изменение было слишком *объемным*, затрагивающим 8721 фрагмент программы. Но должен же быть способ, позволяющий разбить это изменение на более мелкие части.

Что если попробовать разбить его на 8721 более мелкое изменение, каждое из которых относилось бы к одному фрагменту программы? Тогда я смог бы брать и тестирувать одно изменение за другим до тех пор, пока тест не даст сбой и последнее взятое изменение будет тем самым, не прошедшим тестирование. Иными словами, я смоделировал бы историю разработки версии 4.17. (Вообще-то сам я моделировать историю не собирался, этим должно было заняться созданное мной инструментальное средство. И пока я попивал чай, играл с детьми или просматривал электронную почту, этот изящный и несложный инструмент вел бы поиск и обнаружение дефектных изменений. И все было бы просто замечательно.)

Но меня подстерегала ловушка. У меня не было ключа к последовательности, в которой происходили эти изменения. Это было весьма существенным препят-

ствием, поскольку отдельные изменения могли зависеть друг от друга. Например, изменение A могло ввести переменную, которая использовалась бы в новом коде, включенном в другое изменение, B или C. При каждом обращении к изменению B или C нужно было также обращаться и к изменению A; в противном случае рассыпалась бы вся структура gdb. Примерно так же изменение X могло переименовать некоторые определения функций; это переименование могло бы отразиться на других изменениях (Y, Z). Если обращаться к изменению X, то нужно также обращаться и к изменениям Y и Z, поскольку опять же нарушилась бы конструкция gdb.

Но как определить, зависит какое-нибудь изменение от других или нет? Эта проблема представлялась очень сложной и почти неразрешимой без какого-нибудь необычного (и пока что несуществующего) программного анализа нескольких версий.

А что если просто попытаться воспользоваться различными вариантами порядка внесения изменений? 8721 отдельное изменение может быть последовательно произведено $8721 \times 8,720 \times 8,719 \times \dots \times 1 = 8,721!$ различными способами. Протестировать все эти способы не представляется возможным. Скорее, подошло бы тестирование всех подгрупп: 8721 изменение означает $2^{8,721} = 10^{2,625}$ возможных подгрупп, значит, тестов будет намного меньше, чем при тестировании всех вариантов последовательностей 8721 изменения. Можно, конечно, утешиться мыслью, что к тому времени, когда эти вычисления на моем компьютере завершатся, уже войдут в обиход квантовые компьютеры, путешествия во времени и универсальные программы, исправляющие код, и надобность в моих тщетных попытках отпадет сама собой.

Поэтому я предпринял другую попытку. Может быть, обратиться к старому испытанному способу «разделяй и властвуй»? Сначала можно подвергнуть тестированию первую половину изменений исходного кода gdb 4.16. Если ddd откажет, мы будем знать, что дефектные изменения были в этой первой половине, а если отказа не будет, мы продолжим поиск в другой половине. С каждым тестом мы будем сокращать пространство поиска наполовину и таким образом остановимся на дефектном изменении. Это именно то, о чем я думал: автоматическое применение научного метода, которое систематически выдвигает, тестирует и уточняет гипотезу.

Но опять-таки, что делать, если используемая последовательность изменений завершается в непоследовательном участке кода? У меня на этот счет не было никаких идей.

Автоматическое обнаружение причин отказа

На решение, которое пришло мне в голову совершенно случайно в шесть часов утра, когда я еще лежал в постели, ушло три месяца. Солнце вставало, птицы пели, и я наконец-то додумался. Мои рассуждения были следующими.

О Использование половины изменений дает весьма скромный шанс получения последовательной конструкции — слишком высок риск пропустить зависимое

изменение. Но если взять последовательное построение (или уже «готовый» результат), можно очень быстро сузить круг изменений.

- С другой стороны, использование *отдельных* изменений имеет намного больше шансов получить что-либо существенное, в частности, если измененная версия уже была последовательной. К примеру, представьте себе изменение отдельной функции; пока ее интерфейс не изменится, мы, скорее всего, получим работоспособную программу. Но проверка одного за другим всех изменений займет целую вечность.

Поэтому я пошел на компромисс: начну с двух половин. Если ни одна из этих половин изменений не будет тестируемой конструкцией, то я вместо этого разобью последовательность изменений на *четыре* подгруппы, а затем использую каждую подгруппу отдельно с исходным кодом `gdb 4.16`. Кроме этого, я также *уберу* подгруппу из исходного кода `gdb 4.17` (что будет реализовано за счет использования подмножества, целиком извлеченного из исходного кода `gdb 4.16`).

Разбиение на четыре (вместо двух) означает, что будет использована *меньшая* последовательность изменений, значит, измененные версии были бы ближе к исходным (работоспособным) версиям, что подразумевает более высокие шансы получения последовательной конструкции.

Если четырех подгрупп будет недостаточно, то я перейду к 8, 16, 32 и т. д., пока в конечном итоге не стану использовать каждое отдельное изменение одно за другим, что даст мне наивысший шанс получения последовательной конструкции. Как только я получу тестируемую конструкцию, алгоритм будет запущен сначала.

Я подсчитал, что в самом худшем случае алгоритм потребует для своей реализации $8721^2 = 76\,055\,841$ тестов. Это все еще слишком большое количество, но все же намного меньшее, чем при экспоненциальных подходах, рассмотренных ранее. Если взять другую крайность, при которой все конструкции подойдут для тестирования, алгоритм будет работать как двоичный поиск, и потребует всего лишь $\log_2 8721 = 14$ тестов. Стоило ли этим заниматься с учетом разницы?

Я создал простой сценарий на языке Python с весьма приблизительной версией предыдущего алгоритма. Его ключевым фрагментом была функция тестирования. Он должен был брать последовательность изменений, запускать `patch` для вставки изменений в исходный код `gdb 4.16`, а затем вызывать `make` для создания измененного `gdb`. В завершение он должен был запускать `gdb` и следить за возникновением отказа (возвращая «`fail`») или прохождением теста (возвращая «`pass`»). Поскольку любой из этих шагов мог потерпеть неудачу, тестирующая функция могла также вернуть в качестве результата «`unresolved`» (решение не принято).

Как только я запустил сценарий, тут же оказалось, что чаще всего возвращалось значение «`unresolved`». Фактически примерно для 800 первых тестов функция тестирования вернула лишь «`unresolved`». Количество последовательностей изменений возросло с двух до четырех, потом до восьми... пока у нас не получилось 64 подгруппы, каждая из которых содержала 136 изменений. На выполнение этих тестов ушло некоторое время. Поскольку одно создание `gdb`

занимало около шести минут, то мое ожидание продлилось три дня. (Вообще-то я не сидел без дела, а работал над кандидатской диссертацией. Но тем не менее...)

Не успел я приступить к изучению регистрационного журнала, как случилось нечто необычное. Тест наконец-то выдал отказ! Теперь я должен был увидеть, как алгоритм фокусируется на более мелкой подгруппе, сужая пространство поиска. Но когда я проверил результаты, оказалось, что тест, прежде чем остановиться, вывел на экран следующее сообщение:

```
NameError: name 'next_c_fial' is not defined
```

После трех дней непрерывной работы мой сценарий споткнулся на простой опечатке. И я пожалел, что не использовал вместо Python какой-нибудь язык со статическим контролем.

Я все исправил и запустил тест снова. Теперь-то он должен был заработать. По прошествии еще пяти дней и проведении примерно 1200 тестов сценарий наконец-то выделил дефектное изменение: то самое изменение в коде gdb, вызвавшее отказ ddd. Это было изменение в одной-единственной строке, причем даже не в программном коде, а во встроенным тексте:

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
...
> "Set argument list to give program being debugged when it is started.\n\n
```

Это изменение от arguments (аргументы) к argument list (список аргументов) стало причиной того, что gdb 4.17 больше не работал с ddd. Этот текст выводился отладчиком gdb, когда пользователь обращался за помощью к команде set args. Но он использовался также и в других местах. Когда выдавалась команда show args, gdb 4.16 отвечал:

```
Arguments to give program being debugged is "11 14"
```

а gdb 4.17 давал следующий ответ:

```
Argument list to give program being debugged is "11 14"
```

Именно этот новый ответ и запутывал ddd, поскольку им ожидался ответ, начинающийся со слова Arguments. Таким образом, мой сценарий после пяти дней работы все же нашел дефектное изменение, но при этом нужно учесть, что работала полностью автоматизированная версия.

Дельта-отладка

Несколько следующих месяцев я потратил на усовершенствование и оптимизацию алгоритма и инструментального средства, чтобы в конечном итоге на поиск дефектного изменения тратилось не более часа. Со временем я опубликовал алгоритм под названием *delta debugging* (дельта-отладка), поскольку он вел «отладку» программы путем выделения дельты, или различий между двумя версиями.

В этой главе я покажу свою реализацию алгоритма дельта-отладки на языке Python. Функция `dd()` воспринимает три аргумента — два перечня изменений и тест.

- Перечень `c_pass` содержит «работоспособную» конфигурацию, перечень изменений, которые можно применить, чтобы заставить программу работать. В нашем (вполне типичном) случае — это пустой перечень.
- Перечень `c_fail` содержит «отказавшую» конфигурацию, в нашем случае перечень изменений, вызывающих отказ программы. Это должен быть перечень из 8721 изменения (которые мы могли бы инкапсулировать, скажем, в объектах `Change`).
- Функция тестирования, которая воспринимает перечень изменений, использует эти изменения и запускает тест. В качестве выходных данных она возвращает `PASS`, `FAIL` или `UNRESOLVED`, в зависимости от успешности завершения теста, его отказа или неразрешенного исхода. В нашем случае функция тестирования использует изменения в виде вставки в программу и запускает тест в рассмотренном ранее порядке.
- Функция `dd()` методично сужает различия между `c_pass` и `c_fail` и в конечном итоге возвращает тройственные значения. Первое из этих значений должно быть выделенным различием — `delta`, одиночным объектом `Change`, в котором содержится одностороннее изменение в исходном коде `gdb`.

Если вы собираетесь самостоятельно внедрять `dd()`, то можете свободно воспользоваться показанным здесь (и включенным в посвященный этой книге веб-сайт издательства *O'Reilly*) код. Вам также потребуются три вспомогательные функции:

```
split(перечень, n)
```

Разбивает перечень на `n` составляющих одинаковой длины (за исключением, может быть, последней). Так:

```
split([1, 2, 3, 4, 5], 3)
```

выдает:

```
[[1, 2], [3, 4], [5]]
```

```
listminus() и listunion()
```

Возвращают, соответственно, то, что разъединяет или объединяет две группы, представленные в виде перечня. Так:

```
listminus([1, 2, 3], [1, 2])
```

выдает:

```
[3]
```

тогда как:

```
listunion([1, 2, 3], [3, 4])
```

выдает:

```
[1, 2, 3, 4]
```

Код на языке Python показан в примере 28.1.

Пример 28.1. Реализация алгоритма дельта-отладки

```
def dd(c_pass, c_fail, test):
    """Возвращает троицу (DELTA, C_PASS', C_FAIL') так что
    -C_PASS является эквивалентом подгруппы C_PASS',
    подгруппа C_FAIL'
    эквивалентна содержимому подгруппы C_FAIL'
    -DELTA = C_FAIL' - C_PASS' - это минимальная разница
    между C_PASS' и C_FAIL', существенная для TEST."""
    n = 2 # Количество подгрупп

    while 1:
        assert test(c_pass) == PASS # Константа
        assert test(c_fail) == FAIL # Константа
        assert n >= 2

        delta = listminus(c_fail, c_pass)

        if n > len(delta):
            # Никакого дальнейшего уменьшения
            return (delta, c_pass, c_fail)

        deltas = split(delta, n)
        assert len(deltas) == n

        offset = 0
        j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])

            if test(next_c_fail) == FAIL and n == 2:
                c_fail = next_c_fail
                n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:
                c_pass = next_c_fail
                n = 2; offset = 0; break

            elif test(next_c_pass) == FAIL:
                c_fail = next_c_pass
                n = 2; offset = 0; break
            elif test(next_c_fail) == FAIL:
                c_fail = next_c_fail
                n = max(n - 1, 2); offset = i; break
            elif test(next_c_pass) == PASS:
                c_pass = next_c_pass
                n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1

        if j >= n:
            if n >= len(delta):
                return (delta, c_pass, c_fail)
            else:
                n = min(len(delta), n * 2)
```

Минимизация входных данных

Дельта-отладка (или любая другая система автоматизации научного метода) поражает своей универсальностью. Поиск причин можно вести не только в последовательности изменений, но и в других поисковых пространствах. К примеру, дельта-отладку можно легко приспособить для поиска причин отказа во входных данных программы, что мы с Ральфом Хилдебрантом (Ralf Hildebrandt) и сделали в 2002 году.

При поиске причин во входных данных программный код остается без изменений: к нему не применяются никакие модификации и реконструкции, он только запускается на выполнение. Вместо него изменениям подвергаются входные данные. Представьте себе программу, которая работает с большинством вариантов входных данных, но выдает отказ при работе с каким-то определенным вариантом. С помощью дельта-отладки без особого труда можно выделить дефектные различия между двумя вариантами входных данных: «Причина отказа веб-браузера заключается в теге `<SELECT>` в строке 40».

Можно модифицировать алгоритм, чтобы он возвращал минимизированные входные данные: «Чтобы получить отказ веб-браузера, ему нужно дать в обработку веб-страницу, содержащую тег `<SELECT>`». В минимизированном вводе для поиска причины важен каждый отдельный остающийся символ. Минимизированный ввод может быть очень полезен для отладчиков, упрощая их работу: их рабочий цикл сокращается, а количество исследуемых состояний уменьшается. В качестве немаловажного (и, наверное, красивого) побочного эффекта они фиксируют сущность отказа.

Однажды мне встретились программисты, разбирающиеся в ошибках базы данных от стороннего поставщика. Они использовали очень сложные, генерированные машиной SQL-запросы, которые временами приводили к отказу базы. Поставщик не хотел воспринимать эти отказы всерьез, поскольку «вы единственные наши клиенты, использующие столь сложные запросы». Тогда программисты упростили односторонний SQL-запрос до одной-единственной строки, которая вызывала отказ. После предъявления этой строки поставщику тот немедленно дал ошибке самый высокий приоритет и устранил ее.

Как достичь минимизации? Проще всего предоставить функции `dd()` пустой перечень `c_pass`, чтобы тест считался пройденным и возвращал «pass», только если входные данные пусты, а в противном случае возвращал «unresolved». Перечень `c_pass` оставался без изменений, а `c_fail` с каждым непройденным тестом становился все меньше и меньше.

В свою очередь, все, что требуется для выделения таких причин отказа, — это автоматизированный тест и средство для разбиения входных данных на более мелкие части — то есть функция разбиения, имеющая какие-то базовые понятия о синтаксисе входных данных.

Поиск дефекта

В принципе дельта-отладка может также минимизировать и весь программный код, чтобы иметь в своем распоряжении только то, что относится к делу. Пред-

положим, что ваш веб-браузер отказывает при распечатке HTML-страницы. Применение дельта-отладки к коду программы означает, что должен быть оставлен только тот небольшой фрагмент кода, который требуется для воспроизведения отказа. Неужели непонятно? К сожалению, на практике это вряд ли сработает. Причина в том, что элементы программного кода сильно зависят друг от друга. Извлечешь одну часть — и все рассыплется. Вероятность получения чего-то существенного за счет произвольного извлечения частей крайне мала. Поэтому дельта-отладка почти наверняка потребовала квадратичного числа тестов. Даже для программы, состоящей из 1000 строк, это уже будет означать миллион тестов и многие годы ожиданий. Мы не располагаем стольким количеством времени, поэтому к подобному подвигу не готовы.

Тем не менее нам нужно отловить причину отказа не только во входных данных или в последовательности изменений, но и в самом исходном коде — иными словами, нам нужны операторы, вызвавшие отказ программы. (И разумеется, мы хотим их получить в автоматическом режиме.)

И эта задача оказалась по плечу дельта-отладке. Но решения «в лоб» не получилось. Нам понадобилось совершить обходной маневр и воспользоваться состоянием программы — то есть набором всех программных переменных и их значений. Мы хотели автоматически определить причины отказа в этом наборе, получая сообщения вроде: «При обращении к `shell_sort()` переменная `size` вызывает отказ». Но как этого добиться?

Давайте подведем итог всему ранее сделанному. Мы сделали дельта-отладку, работающую с версиями программы — рабочей и нерабочей, — и выделили минимальные отличия, ставшие причиной отказа. Мы сделали дельта-отладку входных данных программы — опять-таки рабочих и нерабочих — и выделили минимальные отличия, ставшие причиной отказа. Если мы применим дельта-отладку к состояниям программы, то возьмем ее состояние при работающем запуске и состояние программы при неработающем запуске, а в итоге получим минимальную разницу, вызывающую отказ.

Но здесь у нас возникают три проблемы. Проблема номер один: *Как получить состояние программы?* Получается, сначала мне нужно оборудовать отладчик `gdb` запросом всех имен переменных, а затем *раскрыть* все структуры данных. Если попадется массив или структура, мне нужно запросить входящие туда элементы; если обнаружится указатель, мне нужно запросить переменную, на которую он указывает, и т. д. — до тех пор, пока не будет достигнута конечная точка или не получен набор всех доступных переменных. Если абстрагироваться от конкретных адресов памяти, это состояние программы может быть представлено в виде графа переменных (вершин) и ссылок (ребер), как показано на рис. 28.2.

Следующая проблема: *Как сравнить два состояния программы?* Это оказалось относительно простой задачей: существуют известные алгоритмы вычисления общих подграфов между двумя графами — и все, что не является частью подграфа, становится различием. Теперь при правильном использовании такого алгоритма мы сможем сделать выборку и определить различия в двух состояниях программы.

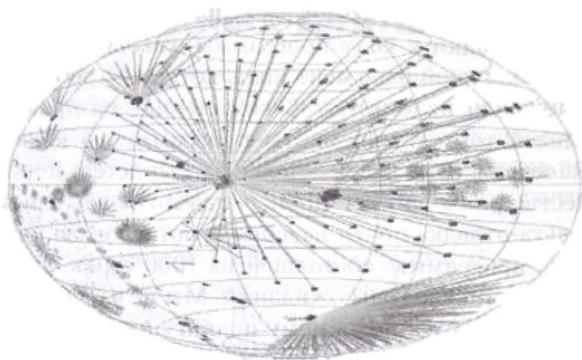


Рис. 28.2. Состояние программы-компилятора GNU

Третья и последняя проблема: *Как определить различия между состояниями программы?* Это была непростая задача, поскольку в ней задействовалось не только наблюдение, но и воздействие на состояния программы. Для использования различий в состоянии программы нам нужно было присвоить переменным новые значения, но кроме этого, еще и повторить всю сложную структуру данных, включая распределение и удаление элементов. Как только это получится, дело пойдет веселее; мы сможем передавать состояния программы между запущенными процессами. И не только полное состояние программы, но и ее частичные состояния — начиная с небольших изменений в отдельной переменной и заканчивая крупномасштабными изменениями, равными, скажем, половине таблицы имен.

К этой идеи переноса состояний программы при ее выполнении нужно было еще привыкнуть. Припоминаю первую презентацию в IBM, где я объяснил алгоритм, его применение к состояниям и перешел к завершающему примеру: «Теперь у нас есть 879 различий между этими двумя состояниями. Позволим дельта-отладке сузить область, содержащую причину отказа. На этой стадии алгоритм берет половину различий, то есть 439 различий состояния, и работает с ними. Это означает, что теперь в работоспособном запуске для 439 переменных устанавливаются значения, обнаруженные в неработоспособном запуске...»

И в этот момент ко мне подходит парень из зала и говорит: «По-моему, все это пустая затея».

Разумеется, он был прав. Никакой существенной пользы из установки 439 переменных значений, используемых в другом запуске программы, не было, как и не было пользы от установки значений другим 440 переменных. Но именно в этой ситуации на первый план выступает дельта-отладка с идеей создания меньших изменений — то есть будут опробованы 220 переменных, 110 и т. д. В конечном итоге будет выделена переменная, вызвавшая отказ: «Отказ компилятора был вызван циклом в абстрактном синтаксическом дереве». И такое завершение, разумеется, оправдывает средства — в частности, для сотрудников IBM, которые в плотную занимались разработкой (и отладкой) компиляторов.

Таким образом, доказательство работоспособности помогло людям забыть о странностях такого подхода. Но моя первая публикация по этой теме была принята далеко не сразу. Один рецензент искренне признался, что был настолько удивлен необычностью подхода, что даже не потрудился дочитать до описания результатов.

Тем не менее обнаружение причины отказа в состоянии программы было только обходным маневром на пути к завершению всего решения. Точку в этой технологии поставил Холгер Клив (Holger Cleve). Так как он знал, какие переменные вызывают отказ, оставалось только проследить в обратном порядке их значения до тех операторов, которые их устанавливали — и дело в шляпе! Мы придем к операторам, вызвавшим отказ: «Оператор в строке 437 вызывает цикл, который в свою очередь приводит к отказу». Вот теперь это было настоящим волшебством — и с публикацией этой статьи проблем не возникло.

Но почему люди до сих пор занимаются интерактивной отладкой, когда есть полноценное решение, позволяющее вести ее в автоматическом режиме? Почему мы не вышли на открытый рынок и не стали миллионерами с нашей автоматической отладкой?

Проблема прототипа

Между тем, что можно сделать в лабораторных условиях, и тем, что можно выпустить в виде конечной продукции, есть весьма существенная разница. Основной загвоздкой в нашем подходе была его хрупкость. Он был очень хрупок. Извлечение точного состояния программы — задача не из простых. Предположим, что работа ведется над программой, написанной на языке Си, которая только что была остановлена в отладчике. Вы обнаружили указатель. Указывает ли он на что-нибудь? Если да, то какой из имеющихся в Си тип данных у той переменной, на которую он указывает? И на сколько элементов он указывает? В Си все это остается на усмотрение программиста — и очень трудно догадаться, какое управление памятью использовано в исследуемой программе.

Другая проблема заключается в определении, где заканчивается состояние программы и начинается состояние системы. Некоторое состояние совместно используется несколькими приложениями или приложениями и системой. Где нужно остановить извлечение и сравнение?

Для проведения лабораторных экспериментов эти проблемы можно было рассмотреть и обоснить, но для законченного, полноценного промышленного подхода мы посчитали их непреодолимыми. Именно поэтому люди по-прежнему вынуждены пользоваться интерактивными отладчиками.

Но будущее представляется не таким уж мрачным. Есть доступные инструментальные средства командной строки, выполняющие дельта-отладку входных данных. Дополнительный модуль `ddchange` для Eclipse осуществляет дельта-отладку изменений вашего рабочего стола. Исследователи применяют дельта-отладку к вызовам методов, успешно интегрируя фиксацию и повторный вызов с минимизацией совокупности тестовых данных. И наконец, благодаря всем этим автоматизированным подходам мы получили более глубокое

понимание того, как работает отладка и как она может проводиться систематизированным, иногда даже автоматическим способом, который, как мы надеемся, является самым эффективным и, возможно, самым «красивым».

Вывод

Если вам придется заниматься отладкой, постарайтесь сделать эту работу по возможности менее болезненной. Немалую помощь в этом вам окажет применение системного подхода (и следование научному методу). А автоматизация научного метода поможет еще больше. Но лучшее, что вы можете сделать, — это приложить все усилия к программному коду и к самому процессу разработки. Следуя советам, изложенным в этой книге, вы напишите красивый код — и в качестве побочного эффекта добьетесь также и самой красивой отладки. А какая отладка самая красивая? Разумеется, та, в которой вообще нет необходимости!

Благодарности

Я хочу поблагодарить студентов, с которыми я испытал красоту инструментальных средств отладки. Мартин Бургер (Martin Burger) принял большое участие в создании сервиса AskIgor и реализовал для Eclipse дополнительный модуль ddchange. Холгер Клев (Holger Cleve) занимался исследованиями и реализацией автоматического выделения вызывающих отказ операторов. Ральф Гильдебрандт (Ralf Hildebrandt) реализовал выделение входных данных, вызывающих отказ. Карстен Леманин (Karsten Lehmann) внес свой вклад в AskIgor и реализовал выделение вызывающих отказ состояний программы для Java. Доротея Люткехаус (Dorothea Lutkehaus) написала начальную версию ddd. Томас Циммерман (Thomas Zimmermann) реализовал алгоритм сравнения графов. Кристиан Линдиг (Christian Lindig) и Анжей Васильковски (Andrzej Wasylkowski) дали полезные комментарии к ранним наброскам этой главы.

Дополнительная информация

Я собрал свой опыт систематизированной и автоматической отладки в университете курсе. Там вы можете получить дополнительные сведения о научном методе и дельта-отладке, а также о множестве дополнительных технологий отладки и анализа, таких как статистическая отладка, автоматизированное тестирование или обнаружение статических дефектов. Все лекционные слайды и справочная информация доступны на веб-сайте <http://www.whypartofprogramming.com>.

Если вы особо интересуетесь научными публикациями моей группы, обратитесь к домашней странице дельта-отладки: <http://www.st.cs.uni-sb.de/dd>.

И наконец, поиск в Интернете по ключевой фразе «delta debugging» укажет вам массу источников, включая дальнейшие публикации и разработки.

29 Отношение к коду как к очерку

Юкихиро Мацумото (Yukihiro Matsumoto)¹

Программы в некотором смысле сродни очерку. Про очерк читатели в основном спрашивают: «Чему он посвящен?» А про программу: «Что она делает?» Фактически цель должна быть достаточно ясна, чтобы не вызывать никаких вопросов. Но и для очерка, и для компьютерного кода всегда играет роль то, как они написаны. Даже хороший замысел трудно донести до желаемой аудитории, если он понимается с трудом. Стиль написания не менее важен, чем вынашиваемая при этом цель. Прежде всего и очерк и строки программного кода предназначены для прочтения и понимания человеком.

Вы можете спросить: «Неужели людей можно рассматривать в качестве читателей компьютерных программ?» Предполагается, что люди используют программы для того, чтобы сообщить компьютеру, что нужно делать, а уж затем компьютеры используют компиляторы или интерпретаторы, чтобы оттранслировать и понять код. В конце этого процесса программа транслируется в машинный язык, который обычно читает только центральный процессор. Разумеется, все так и происходит, но это объяснение описывает лишь один аспект компьютерных программ.

Большинство программ не пишется за один присест. За свою жизнь они перерабатываются и переписываются снова и снова. Они должны быть избавлены от ошибок. Изменения технических требований и необходимость повышения функциональных возможностей означают, что сама программа будет постоянно подвергаться изменениям. В течение этого процесса люди должны иметь возможность читать и понимать исходный код; поэтому возможность понимания программы куда более важна для людей, чем для компьютеров.

Разумеется, компьютеры не станут жаловаться на сложность программы, чего нельзя сказать про людей. Код, который сложно прочесть, приведет к существенному снижению продуктивности человеческого труда. А легко читаемый код наоборот будет способствовать ее повышению. И именно в таком коде мы усматриваем красоту.

¹ Эта глава была переведена с японского Невином Томпсоном (Nevin Thompson).

Что придает компьютерной программе свойство хорошей читаемости? Иными словами, каков он, этот красивый код? Хотя у разных людей могут быть разные понятия о том, что такое красивая программа, оценка внешнего оформления компьютерного кода — это вопрос, касающийся не только эстетики. Но вместо этого компьютерные программы оцениваются в соответствии с тем, насколько хорошо они справляются с поставленными перед ними задачами. Иными словами, «красивый код» — это не какое-нибудь абстрактное свойство, существующее независимо от усилий создавших его программистов. Точнее, красивый код на самом деле предназначен для того, чтобы помочь программисту испытать радость и достичь высокой продуктивности в работе. Именно этим я оцениваю красоту программы.

Краткость — один из элементов, помогающих сделать код красивым. Как сказал Пол Грэхэм (Paul Graham): «Лаконичность — это сила». В словаре программирования краткость считается достоинством. Поскольку время, потраченное человеком на просмотр программного кода, тоже имеет свою вполне определенную ценность, в идеале программы не должны содержать ненужной информации.

Например, когда не требуется объявления типа или когда конструкция не требует объявления класса и основной (`main`) подпрограммы, краткость предписывает, что нужно просто воспользоваться такой возможностью и обойтись без всего этого. В качестве иллюстрации этого принципа в примере 29.1 показана программа `Hello World`, написанная на Java и на Ruby.

Пример 29.1. Сравнение кода «Hello World» на Java и на Ruby

Java	Ruby
<code>class Sample {}</code>	<code>print "Hello World\n"</code>
<code>public static void main(String[] argv) { System.out.println("Hello World"); }</code>	

Обе программы выполняют одну и ту же задачу — просто выводят на экран слова «Hello World» — но подходят к этому совершенно по-разному. В Ruby-версии программы нужно всего лишь дать описание самой сути задачи. Отобразить (`print`) «Hello World». Никаких объявлений. Никаких типов данных. А в Java-версии нужно включить массу различных описаний, которые не имеют непосредственной связи с нашими намерениями. Конечно, во всем, что делается в Java при включении всех этих элементов, есть свои определенные преимущества. Но поскольку без всего этого обойтись нельзя, краткость теряется. (В качестве небольшого отступления следует заметить, что Ruby-версия «Hello World» повторяна на трех языках: она также работает на Perl и Python.)

Краткость также может означать избавление от избыточности. Избыточность определяется как дублирование информации. Когда информация дублируется, стоимость постоянной поддержки программы может стать непомерно высокой. А поскольку на постоянную поддержку должен быть затрачен существ-

венный объем времени, избыточность неизбежно снизит продуктивность программирования.

Хотя могут последовать возражения, что избыточность снижает затраты за счет объяснения смысла, на самом деле все обстоит как раз наоборот, поскольку избыточный код содержит слишком много лишней информации. Одним из последствий такой тяжеловесности является то, что избыточный подход зависит от использования инструментальных средств поддержки. Хотя недавно стало популярным при вводе информации полагаться на интегрированные средства разработки (IDE), они не предназначены для помощи в разъяснении сути программы. Настоящий рациональный метод для разработки изящного кода состоит в выборе первоклассного языка программирования. Нужный подход поддерживается Ruby и другими упрощенными языками программирования.

В целях исключения избыточности мы придерживаемся принципа DRY: Don't Repeat Yourself — не повторяйтесь. Если один и тот же код присутствует сразу в нескольких местах, становится непонятным, что именно вы пытались этим выразить.

Концепция DRY является противоположностью программированию типа «скопировать и вставить». В былые времена некоторые организации измеряли продуктивность количеством строк кода, выданных программистом, что фактически приводило к молчаливому поощрению избыточности. Я даже слышал, что копирование как можно большего объема кода выдавалось за достоинство. Но это в корне неверно.

Я верю, что истинное достоинство заключается в краткости. Недавний рост популярности Ruby on Rails состоялся благодаря тому, что в этой среде упорно преследовались краткость и DRY. В языке Ruby придается важное значение принципам «никогда не повторяться» и «соблюдать краткость описаний». Rails унаследовала эту философию у языка Ruby.

Более спорным аспектом красоты кода может быть его узнаваемость. Люди более консервативны, чем вы думаете; большинству из них нелегко освоить новую концепцию или изменить стиль своего мышления. Вместо этого многие предпочитают терпеть все как оно есть, ничего не меняя. Они не хотят менять знакомые им инструментальные средства или изучать новый язык, не имея на то достаточно веских оснований. При любой возможности люди будут сравнивать новые процессы, которые они пытаются изучить, с тем, что они всегда считали проявлением здравого смысла, порой совершенно незаслуженно давая новому процессу отрицательную оценку.

Для смены стиля мышления требуются куда более существенные усилия, чем вы можете себе представить. Чтобы легко перейти к совершенно другой концепции (например с процедурного программирования к логическому или функциональному), необходимо ознакомиться с широким спектром понятий. Нужно поломать над ними голову. Поэтому они снижают продуктивность работы программиста.

Согласно этой точке зрения и в силу того, что Ruby придерживается концепции «красивого кода», — это чрезвычайно консервативный язык программирования. Несмотря на то что он зовется настоящим объектно-ориентированным языком, Ruby не использует прогрессивных управляющих структур, основанных

на передачах вызова метода объекта, как в Smalltalk. Вместо этого Ruby основан на традиционных управляющих структурах, хорошо знакомых программистам, таких как `if`, `while` и т. д. Он даже унаследовал у старого доброго семейства языков Алгол ключевое слово `end`, которым завершаются блоки кода.

По сравнению с другими современными языками программирования, Ruby временами выглядит несколько старомодным. Но важно помнить о том, что «не стоит быть слишком прогрессивным», в этом также заключается секрет написания красивого кода.

Следующий элемент красивого кода — это *простота*. В простом коде часто видится красота. Если программу трудно понять, ее нельзя считать красивой. А когда программа вместо того чтобы быть понятной, написана слишком туманно, это всегда приводит к ошибкам, недочетам и путанице.

Простота — одно из самых недооцененных понятий в программировании. Разработчики языка программирования зачастую стремятся сделать этот язык простым и понятным. При всем благородстве намерений результаты их труда могут привести к усложнению программ, написанных на этом языке. Майк Коулишоу (Mike Cowlishaw), разработавший в IBM язык сценариев Rexx, однажды заметил, что поскольку пользователей языка намного больше, чем их создателей, потребности последних должны уступать потребностям первых:

Все дело в том, что лишь немногим приходится создавать интерпретаторы или компиляторы для языков программирования, а миллионам других людей приходится пользоваться этим языком в повседневной жизни. Поэтому подстраиваться надо под миллионы, а не под единицы. Создатели компиляторов меня за это не любят, поскольку Rexx оказался труден для интерпретации или компиляции, но я считаю, что это оккупится сторицей для людей и, конечно же, для программистов¹.

Я всей душой с ним согласен. Ruby был призван воплотить этот идеал в жизнь, и поскольку он совсем непросто устроен, в нем поддерживаются простые решения в программировании. Отсутствие простоты в Ruby позволяет быть простыми тем программам, в которых он используется. Это также справедливо и для других упрощенных языков программирования; они считаются упрощенными не потому, что реализованы каким-то простым образом, а потому что в них заложено стремление облегчить нагрузку на программиста.

Чтобы посмотреть, что это означает на практике, рассмотрим *Rake*, инструмент конструирования наподобие *make*, широко используемый программистами, работающими на Ruby. В отличие от *Makefile*, которые пишутся в специализированном файловом формате, *Rakefile* пишутся на Ruby, который выступает в качестве своеобразного языка предметной области — Domain Specific Language (DSL), позволяющего заниматься полноценным программированием. В примере 29.2 показан *Rakefile*, запускающий серию тестов.

Пример 29.2. Образец *Rakefile*

```
task :default => [:test]
task :test do
```

¹ Dr. Dobb's Journal, март 1996 г.

```
ruby "test/unittest.rb"
end
```

В *Rakefile* используются преимущества следующих сокращений, допустимых в синтаксисе Ruby.

- о Скобки для аргументов метода могут быть опущены.
- о Хэш-пары ключ–значения могут появляться в конце методов без скобок.
- о Блоки кода могут присоединяться к окончанию вызовов методов.

Программировать на Ruby можно и без использования этих синтаксических элементов, поэтому теоретически они излишни. Их использование часто критируется за то, что они усложняют язык. Впрочем, в примере 29.3 показано, как код примера 29.2 был бы написан без использования этих особенностей.

Пример 29.3. Rakefile без использования сокращенного синтаксиса

```
task({:default => [:test]})
task(:test, &lambda() {
    ruby "test/unittest.rb"
})
```

Вполне очевидно, что если бы синтаксис Ruby был лишен сокращений, язык стал бы более изысканным, но у программистов прибавилось бы работы, а их программы стало бы труднее читать. Поэтому когда простые инструменты используются для решения сложной проблемы, сложность просто перекладывается на плечи программиста, а это фактически означает то же, что и ставить телегу впереди лошади.

Другим важным элементом концепции «красивого кода» является гибкость. Я определяю гибкость как *свободу от вынужденного использования инструментальных средств*. Когда программисты вынуждены делать что-то против своей воли, ради выполнения требований инструментальных средств, это вызывает их раздражение. Такое давление негативно оказывается на программисте. Конечный результат далек от радостных впечатлений и от понятий красоты, соответствующих нашим определениям красоты кода. Люди представляют намного большую ценность, чем любые инструментальные средства или языки. Компьютеры должны служить программистам для придания их труду наибольшей производительности и радости, но в действительности они довольно часто увеличивают нагрузку, вместо того чтобы ее уменьшать.

Сбалансированность является заключительным элементом красоты кода. До сих пор я вел речь о краткости, консерватизме, простоте и гибкости. Но сами по себе эти элементы не гарантируют получения красивой программы. Каждый элемент будет работать в гармонии с другими на создание красивого кода, если будет с ними сбалансирован и учтен с самого начала работы над программой. А если вдобавок к этому вы поймете, что испытываете удовольствие от создания и чтения кода, значит, вы, как программист, счастливы.

Счастливого программирования!

30 Когда кнопка остается единственным предметом, связывающим вас с внешним миром

Арун Мехта (Arun Mehta)

«Профессор Стивен Хокинг (Stephen Hawking) в состоянии лишь нажать кнопку» — мы получили именно такое односточное техническое задание.

Выдающийся физик-теоретик профессор Хокинг страдал боковым амиотрофическим склерозом (ALS). Этот недуг «характеризуется постепенным перерождением клеток центральной нервной системы, управляющих осознанным движением мышц. Это расстройство вызывает мышечную слабость и атрофию всего тела»¹. Он вел записи и переговоры, используя программную систему Equalizer, которой управлял посредством одной-единственной кнопки. Для преобразования текста в речь он использовал уже снятую с производства внешнюю аппаратуру. Исходный код Equalizer также был утрачен.

Чтобы не терять работоспособности в случае отказа его устаревшего оборудования, он обратился в некоторые компании по разработке программного обеспечения с заказом на создание программы, позволяющей человеку с существенными ограничениями двигательных функций иметь доступ к компьютерной технике. Компания Radiophony, основанная мной совместно с Викрамом Кришна (Vickram Crishna) с охотой взялась за решение этой задачи. Мы назвали программу eLocutor² и решили сделать ее общедоступной, с открытым кодом, чтобы не было повторения проблем с Equalizer.

Важность таких программ для жизни людей с физическими ограничениями трудно переоценить. История профессора Хокинга, безусловно, была наилучшим подтверждением этому. Он получил возможность стать не только одним из наших ведущих ученых, но также и необычайно успешным автором и популяризатором, только лишь потому, что программное обеспечение позволяло

¹ http://en.wikipedia.org/wiki/Amyotrophic_lateral_sclerosis.

² Веб-сайт загрузки <http://holisticit.com/eLocutor/elocutor3.htm>.

ему составлять тексты и говорить. Кто знает, сколько гениев осталось в безызвестности только потому, что ребенок не мог говорить или достаточно понятно для учителей излагать свои мысли на бумаге.

Профессор Хокинг продолжал пользоваться программой Equalizer, с которой он был знаком в течение нескольких десятилетий. Тем временем eLocutor оказалась полезна для людей с различными формами расстройств благодаря, в частности, своей легкой настраиваемости под человеческие нужды.

Один из первых вопросов, ставших предметом объяснения этой проблемы любому конструктору, звучал так: неужели мы не можем найти способ увеличения количества средств ввода, доступных профессору Хокингу? Но его помощник не сдавался: Equalizer работал с помощью единственной кнопки, и они не видели смысла что-либо менять. Мы также видели здравый смысл в создании программного обеспечения для людей с самыми исключительными случаями физических расстройств, поскольку существовало множество разновидностей двоичных переключателей, которые даже самый беспомощный человек мог нажимать, приводить в действие движением плеча, брови или языка или задействовать за счет непосредственной мозговой активности¹. Разработав решение, доступное множеству людей, мы могли следующим шагом продумать вопрос ускорения ввода для тех, кто был проворнее остальных.

Мы также рассматривали рыночную нишу, позволяющую адаптировать eLocutor к более широкому кругу пользователей. Программы, управляемые одной кнопкой, вполне могли бы подойти, к примеру, для мобильных телефонов: гарнитуры, освобождающие от ручного управления, имеют всего лишь одну кнопку. Наряду с соответствующим текстово-речевым преобразованием, позволяющим оторвать взгляд от экрана, такой комплекс подошел бы также и водителям автомашин. Или можно представить себе иной сценарий: на встрече с клиентом не отрывая от него глаз можно будет организовать на Google поиск упомянутого собеседником имени и незаметно получить озвученные результаты.

Разумеется для разработчика программного обеспечения изобретение редактора, эффективно работающего при помощи одной-единственной кнопки, было очень интересной технической задачей. Сначала нужно было определить основной круг задач, выполняемых программой eLocutor. Мы выбрали извлечение и хранение файла, распечатку, удаление, озвучку, прокрутку и поиск.

Затем нам пришлось найти способы осуществления всех этих действий при помощи одной-единственной кнопки. Это была одна из самых захватывающих частей задачи, поскольку программист нечасто принимается за работу на уровне разработки основных принципов управления информацией. Именно этой деятельности и посвящена основная часть повествования данной главы.

Основная модель конструкции

Излишне упоминать о том, что программы должны работать эффективно, чтобы пользователь мог вести быстрый набор текста без слишком частых щелчков кнопкой. Порой на набор одного слова у профессора Хокинга уходили минуты,

¹ См., к примеру, <http://www.brainfingers.com/>.

поэтому любое ускорение редактирования было бы благом для занятого человека.

Несомненно, программа нуждалась в гибкости настройки. Особенности и размеры словаря наших пользователей могли существенно различаться. Программе нужно было уметь приспосабливаться к этому. И потом, у нас было очень большое желание предоставить человеку с физическими расстройствами возможность вносить изменения во многие установки и конфигурации по возможности самостоятельно, не прибегая к посторонней помощи.

Поскольку мы не достигли значительных подвигов в разработке технических условий и не имели достаточного опыта в написании такого рода программного обеспечения, мы были готовы к весьма существенным поправкам к конструкции по мере роста нашего понимания решаемой задачи. Учитывая все эти условия, мы решили создавать программу на Visual Basic версии 6, пре-восходном скоростном инструменте создания прототипов, имеющем большое разнообразие готовых элементов управления. VB упрощал создание графического пользователяского интерфейса и обеспечивал удобный доступ к свойствам базы данных.

Особенностью данной задачи была необычно высокая асимметрия в потоках данных. Пользователь с достаточно хорошим зрением должен иметь большую возможность для восприятия информации. Но для людей с существенными нарушениями двигательной активности поток данных в обратном направлении был крайне незначителен: всего лишь какой-нибудь редкий бит.

Программа предлагала пользователю поочередный выбор, а он щелчком принимал предлагаемый вариант, если тот ему подходил. Разумеется проблема была в том, что для любого места существовало множество вариантов. Пользователю могло понадобиться ввести любой из десятков символов, или сохранить, прокрутить, найти или удалить текст. Если бы eLocutor осуществлял циклический перебор всех вариантов, это заняло бы много времени, поэтому варианты в нем были сведены в группы и подгруппы, структурированные наподобие дерева.

Чтобы ускорить ввод, eLocutor заглядывал вперед, предлагая способы завершения вводимого слова и варианты следующего слова, а также окончание всей фразы. Пользователь должен был заранее знать об этих догадках и иметь возможность определить доступные ему сокращения.

Поэтому мы решили создать визуальный интерфейс, в котором происходило бы динамическое изменение размеров элементов или даже их исчезновение, в зависимости от наших предположений о желаемом пользователем информационном отображении, с целью предоставить сокращения, помогающие быстро разгадать желаемое предложение. Поэтому при пользовательском вводе экран eLocutor содержит окно с предложениями, как можно было бы завершить данное слово, и другое окно, которое помогает выбрать следующее слово. (Группы символов пунктуации тоже рассматриваются как слова.) Если начало набираемого предложения было идентично каким-нибудь предложениям, хранящимся в базе данных, они также отображаются на экране. На рис. 30.1 показан типовой экран eLocutor.

Иногда вариантов так много, что они не помещаются в небольшом окне. Свойство сканирования помогает пользователю быстро выбрать нужный из них. Оно открывает большое окно, показывая все варианты, а более мелкие группы из их числа последовательно появляются в меньшем по размеру окне. Слово, появляющееся в большом окне информирует пользователя, что eLocutor может предложить ему сокращение для его набора. Теперь он ждет его появления в малом окне, после того как производит щелчок. Большое окно исчезает, и теперь варианты из малого окна, которых около десятка, становятся доступными пользователю как обычно, посредством дерева.



Рис. 30.1. Экран eLocutor

Полезная площадь экрана заново разбивается на зоны, когда пользователь прекращает набор текста и начинает его прокручивать, в это время на экране показывается максимально возможный объем текста до и после места вставки.

Прогноз должен быть максимально разумным, чтобы наилучшим образом использовать щелчки, которые пользователю с физическими расстройствами даются с трудом. Интеллектуальная обработка,строенная нами в программу, имеет три характерные особенности.

Реляционная база данных

Когда пользователь вводит первые несколько символов слова, осуществляется поиск в словарной таблице, предоставляющий предложения по его завершению. Анализ предыдущего введенного пользователем текста также показывает, какое следующее слово он может выбрать.

Кэш

Он позволяет извлечь преимущества из типовых моментов человеческого поведения. Мы кэшируем не только часто используемые слова, но и имена файлов, условия поиска, разговорный текст и пути принятия решения, чтобы пользователь мог без труда воспроизвести последовательность шагов.

Специальные группировки

Эта составляющая позволяет извлечь преимущества из естественной группировки слов, таких как названия городов, продуктов, частей речи и т. п. Такие группировки позволяют пользователю выстраивать новые предложения на основе старых за счет быстрой замены слов в часто употребляемых фразах на другие, подобные им слова. Например, если предложение: «Пожалуйста, принеси мне соль» имеется в базе данных, несколько щелчков позволяют составить предложение: «Пожалуйста, принеси мне сахар».

Объединение всех доступных вариантов представлено в виде дерева, похожего на иерархическое меню. Варианты, представленные в дереве, выделяются друг за другом, периодически сменяясь в постоянном темпе. Структура дерева также естественно разбивается на поднаборы вариантов, подобные только что рассмотренным группам слов.

Различные составные части экрана, показанного на рис. 30.1, нуждаются в пояснении. Активная часть редактируемого пользователем текста показана в среднем окне, а содержимое окон над и под ним подстраивается под действия пользователя. Справа и снизу располагаются предположения, которые делает программа относительно того, что вы можете захотеть ввести дальше.

Текст в правом верхнем углу (выделенный красным цветом на пользовательском экране) состоит из предложений по заменам последнего слова на тот случай, если вы ввели несколько символов слова и хотите, чтобы eLocutor догадался о его продолжении. Если вы завершили ввод последнего слова, то чуть ниже в черном цвете представлены предложения для следующего слова. Группы символов пунктуации тоже рассматриваются как слова, и так как последнее слово состоит из буквенно-цифровых символов, следующее слово может быть символами пунктуации, как показано справа на рис. 30.1.

Когда пользователь набирает предложения, похожие на те, что уже были им набраны ранее, он может воспользоваться предложениями, которые показаны внизу экрана. Но в основном внимание пользователя привлечено к дереву слева, являющемуся единственным средством извлечения пользы из всей предлагаемой информации для оказания влияния на текст, расположенный в среднем окне.

Чуть ниже дерева пользователь может увидеть, сколько вариантов различного типа ему доступно, а также другую полезную информацию, которую мы рассмотрим чуть позже.

Интерфейс осуществляет последовательный проход по элементам дерева. Имея в своем распоряжении одну кнопку, пользователь нажимает ее в нужный момент, когда выделен элемент, который он хочет выбрать. Различные окна экрана показывают пользователю варианты, доступные для следующего слова, для завершения слова, для завершения фразы и т. д. Чтобы воспользоваться

этими вариантами, он должен ждать перемещения, пока соответствующий вариант не будет ему предложен в дереве меню.

Интерфейс ввода

В качестве единственного двоичного средства ввода мы выбрали правую кнопку мыши. Это позволяет без особого труда подключить к eLocutor множество разнообразных кнопок. Любой электротехник или радиолюбитель может создать такое подключение, вскрыв мышь и припаяв желаемую кнопку параллельно ее правой кнопке.

На рис. 30.2 показано, как мы создали временное подключение к специальному переключателю профессора Хокинга: монтажная плата слева внизу взята из внутренностей мыши, а внешний переключатель припаян к точкам подключения правой кнопки.

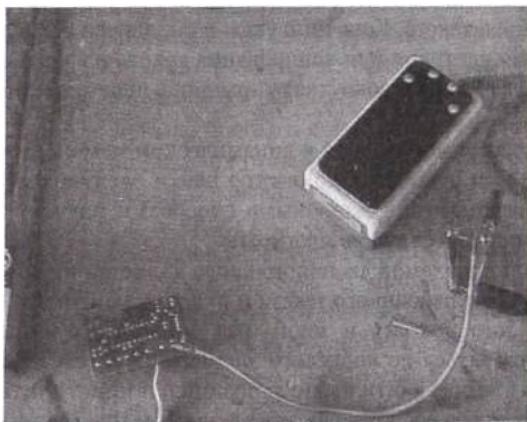


Рис. 30.2. Подключение переключателя профессора Хокинга параллельно правой кнопке мыши

Дерево

Если вы можете предоставить программе всего лишь единственный двоичный ввод, то наличие одной составляющей графического пользовательского интерфейса вполне очевидно: все выбираемые варианты должны быть представлены друг за другом в форме двоичного дерева. Каждый узел выбирается интерфейсом, если пользователь нажал кнопку в заданный промежуток времени, при этом могут быть открыты дополнительные варианты в виде поддерева. Если пользователь не нажал кнопку, программа автоматически перемещается на следующий дочерний узел и снова ждет нажатия кнопки.

Для реализации дерева мы воспользовались элементом управления Visual Basic TreeView¹. Его нужно рассматривать в качестве дерева, растущего слева направо. Если на любом узле будет произведен щелчок в выбранный пользователем интервал времени, который устанавливается с использованием элемента управления таймером — Timer control, вы развернете узел и подниметесь вверх по дереву (то есть переместитесь вправо) или, если вы находитесь на конечном узле, то будет выполнено какое-нибудь действие. Если щелчок не будет сделан, eLocutor переместит свой фокус к следующему дочернему элементу узла. Если при достижении нижней части не будет нажата кнопка, eLocutor снова начнет перемещение с самого верхнего узла.

Мы заполнили дерево таким образом, что оно предоставляет на каждом уровне дерева узел под названием Up (Вверх), выбор которого перемещает выделение на его родительский элемент, на один уровень ближе к корневому элементу.

Узлами верхнего уровня являются Type (Набор), Scroll (Прокрутка), Edit (Редактирование) (основные функции редактирования) и Commands (Команды) (разнообразные команды). Конечные узлы в поддереве Type (Набор) вводят текст в буфер набора текста. Конечные узлы в поддереве Edit (Редактирование) предназначены для удаления или копирования текста из этого буфера, а конечные узлы поддерева Scroll (Прокрутка) управляют перемещением текста между буферами.

Интеллект eLocutor выражается в динамическом изменении заполнения дерева, чтобы вы могли относительно быстро найти следующее желаемое действие: программа несколькими различными способами изучает ваше поведение, чтобы лучше составлять будущие прогнозы.

Самой большой проблемой двоичного ввода является перемещение. Если вы находитесь в середине набранного текста и нужно удалить что-нибудь в начале предложения, нужно подолгу и много раз дожидаться конечных узлов Up (Вверх), чтобы добраться до корневого элемента, а затем перейти на элемент Scroll (Прокрутка), чтобы найти нужную позицию и приступить к удалению, затем опять несколько раз воспользоваться элементом Up (Вверх), чтобы добраться до корневого элемента, затем перейти к элементу Edit (Редактирование), чтобы выполнить удаление, затем перемещаться снова вверх-вниз по дереву, чтобы прокрутить текст к его окончанию и вернуться к набору. Когда удалось найти ответ на эту дилемму, мы вздохнули с явным облегчением.

Длинный щелчок

Наблюдая за работой принадлежащей профессору Хокингу системы Equalizer, я обнаружил новый режим работы: кроме простого щелчка кнопкой, он мог удерживать ее в нажатом состоянии и отпускать в какой-то ключевой момент. Получалось, что кнопка была уже не просто двоичным, но и аналоговым средством ввода, поскольку могла выдавать сигнал различной продолжительности.

¹ <http://www.virtualsplat.com/tips/visual-basic-treeview-control.asp>.

Мы долго и упорно думали, как наилучшим образом применить эту новую представившуюся нам возможность: теперь из щелчка мы могли извлечь больше информации, чем содержится в простом бите. Мы могли, к примеру, позволить пользователю делать выбор из списка вариантов. Короткий щелчок теперь мог использоваться для исходных действий, а длинный делал доступными множество других вариантов.

Неудивительно, что нам захотелось воспользоваться этим вновьобретенным средством для ряда дополнительных вариантов быстрых переходов. Мы также радовались возможности выполнения различных операций над текстом, выделенным в дереве, например возможности его ввода, копирования в фильтр и т. д. Без использования длинного щелчка мы были ограничены одним действием над конечным узлом, а теперь мы могли предложить пользователю другие возможные варианты относительно того, что делать с выделенным узлом дерева, который необязательно должен быть только конечным узлом.

Перечень дополнительных вариантов выбора не должен быть слишком большим, иначе пользователю придется удерживать кнопку относительно долго. Поэтому мы захотели, чтобы эти варианты выбора изменялись в зависимости от того места дерева, где мы находились. К примеру, элемент «Набрать это» не имел бы смысла, находясь мы в поддереве *Scroll* (Прокрутка), но он был бы весьма кстати в поддереве *Speller* (Орфографический справочник).

То что мы придумали, было простым и понятным режимом работы. Щелчок на узле приводил к выполнению его исходного действия. Но если удерживать кнопку в нажатом состоянии, открывалось отдельное меню, чьи элементы проходили один за другим, и один из них можно было выбрать, отпуская кнопку в тот момент, когда показывался желаемый вариант. Мы использовали этот режим, несколько похожий на щелчок правой кнопкой мыши в Microsoft Windows, чтобы предоставить пользователю контекстно-зависимые элементы меню. Обычно в них включался переход в корневой узел дерева, обратный обход и т. д.

Важность этого дополнительного режима работы трудно было переоценить: существенно повысилась не только скорость ввода и исправления, но и представилась возможность потрясающей гибкости для разработчиков.

Требовалось изящное решение, чтобы сделать меню длинного щелчка контекстно- зависимым, поскольку было слишком обременительно создавать такое специальное меню для каждого узла двоичного дерева. Как и дерево, меню длинного щелчка хранились в виде текстовых файлов, которые можно было подвергнуть редактированию в eLocutor. При выборе соответствующего меню длинного щелчка eLocutor определял, какой узел выделен в данный момент. Если в каталоге меню длинного щелчка существовал текстовый файл под таким же именем, что и имя узла, то в качестве меню брался именно он. Если такого файла не существовало, eLocutor искал имя узла на один уровень выше по дереву и т. д.

Таким образом, каждое поддерево могло иметь свое собственное меню длинного щелчка, всецело находящееся под управлением пользователя. Если представить эту конструкцию по-другому, то можно сказать, что пока не

будет выбрано дочернее меню для замены меню длинного щелчка своего родителя, дочерний узел будет автоматически наследовать родительское меню.

Часть кода реализации длинного щелчка показана в примере 30.1. Функция OpenLongClickFile ищет и открывает файл с именем, аналогичным имени переданного ей параметра, и если он не найден, рекурсивно ищет другой файл, с именем его родителя. Когда истекает время таймера длинного щелчка (long-click timer), в текстовом поле tblongclick отображается новая строка из этого файла. Когда кнопка отпускается, выбирается команда из поля tblongclick. В зависимости от того насколько долго удерживалась кнопка, таймер длинного щелчка запускается повторно. Каждое истечение времени таймера приводит к тому, что код в примере 30.1 проверяет и устанавливает значение булевой переменной ThisIsALongClick, а затем выполняет некоторый фрагмент кода, который необходим, чтобы при каждом длинном щелчке однократно запустить выбор и открытие для чтения соответствующего файла длинного щелчка.

Фрагмент, повторяющийся при каждом истечении времени таймера длинного щелчка, считывает строку из файла и отображает ее в текстовом поле tblongclick. Когда файл закончится, он закрывается и заново открывается, и происходит считывание первой строки. Когда кнопка отпускается, переменная ThisIsALongClick переустанавливается.

Пример 30.1. Реализация контекстно-зависимого меню выбора при длинном щелчке

```
Private Sub longclick_Timer()
Dim st As String
Dim filenum As Long
If Not ThisIsALongClick Then
    ThisIsALongClick = True
    If MenuTree.SelectedItem.Text = stStart Then
        'мы уже в корневом узле
        OpenLongClickFile MenuTree.SelectedItem
    Else
        OpenLongClickFile MenuTree.SelectedItem.Parent
    End If
    If EOF(longclickfilenum) Then
        ' поиск списка элементов меню длинного щелчка, подходящего для данного контекста
        End If
    End If
    Open stlongclickfilename For Input As #longclickfilenum
    Line Input #longclickfilenum, st
    tblongclick = st
End Sub
```

Команды, которые становятся доступными по длинному щелчку, включают в себя:

>Start (В начало)

Осуществляет переход на корневой узел дерева (символ > служит признаком перехода).

Upwards (Вверх)

Перемещает курсор вперед и назад по дереву до тех пор, пока правая кнопка мыши удерживается нажатой. Полезна, если при выделении нужного элемента меню кнопка не будет нажата — то есть если вы пропустите свою очередь.

Type This (Набрать это)

Вводит в среднее окно тот элемент, который выделен в дереве. Доступен только в поддереве Type (Набор).

Set Filter (Установить фильтр)

Копирует выделенный элемент дерева в фильтр; полезен при поиске текста. Также доступен в качестве варианта выбора по длинному щелчку, только если выделенный элемент находится в поддереве Type (Набор).

Words Up, Words Down (Слова вверх, Слова вниз)

Для ускоренной прокрутки в процессе набора, будут рассмотрены чуть позже.

Pause (Пауза)

Полезна, когда команда должна выполняться многократно. Когда пользователь удерживает кнопку нажатой для получения длинного щелчка, перемещение по дереву меню приостанавливается, и один из его элементов находится в выделенном состоянии. Выбор доступного по длинному щелчку элемента Pause (Пауза) поддерживает это состояние приостановки. Теперь при каждом пользовательском щелчке выполняется команда, выделенная в дереве меню. Чтобы выйти из паузы, нужно опять воспользоваться длинным щелчком.

Help (Помощь)

Открывает и проигрывает контекстно-зависимый видеофайл формата .avi, который объясняет, какие варианты выбора из дерева предлагаются пользователю. Подкаталог Help содержит подборку файлов .avi. Он должен содержать как минимум один файл под названием Start.avi. Когда выбирается элемент длинного щелчка Help, проигрывается соответствующий .avi-файл, в зависимости от того, в каком месте дерева меню пользователь находится в данный момент.

Нужный для проигрывания файл ищется по той же схеме, что и меню длинного щелчка. Программа сначала ищет файл с расширением .avi в подкаталоге helpvideos каталога C:\eLocutor. Если такой файл будет найден, он проигрывается; если нет, eLocutor ищет .avi-файл с именем родителя выделенного узла. Если .avi-файл с этим именем не будет найден в каталоге helpvideos, eLocutor рекурсивно восходит по дереву меню до тех пор, пока не найдет узел с соответствующим ему видеофайлом помощи. Это свойство позволяет нам для начала поставлять только обзорный видеофайл и постепенно добавлять все более и более детализированные видеофайлы, которые пользователь должен будет всего лишь скопировать в подкаталог helpvideos, чтобы eLocutor начал их демонстрировать.

Некоторые видеофайлы помощи доступны на веб-сайте <http://www.holisticit.com/eLocutor/helpvideos.zip>. Учитывая динамичную природу этого программного обеспечения, просмотр некоторых видеофайлов поможет читателю быстрее и более основательно разобраться с материалами этой главы.

Динамическое изменение заполнения дерева

Содержимое дерева хранится на диске в виде текстовых файлов. Существенное преимущество такого подхода состоит в возможности динамического редактирования этих файлов, как программой eLocutor, так и пользователем. Иными словами, эти файлы дают нам удобный способ достижения одного из наших конструкторских критериев: дать возможность пользователю самостоятельно приспособить eLocutor к своим нуждам за счет придания структуре данных открытости и легкости редактирования силами пользователя.

Поскольку eLocutor старается прогнозировать, что вы можете захотеть делать дальше, двоичное дерево должно быть динамичным; поддеревья вроде *Next Word* (Следующее слово) часто меняют свое наполнение. Имя каждого файла такое же, как и у узла (с расширением .txt), а содержимое состоит из списка имен его непосредственных дочерних узлов. Если любое из имен узла завершается расширением .txt, значит, они представляют собой корни поддерева, и имена их дочерних узлов могут быть найдены в соответствующем файле. Например, корневой файл называется *Start.txt* и содержит строки *type.txt*, *edit.txt*, *scroll.txt* и *commands.txt*, каждая строка соответствует набору вариантов, отображаемых пользователю в одном из меню, описанном ранее в разделе «Дерево».

Если имя узла не заканчивается на .txt, значит, это конечный узел. Его выбор выражается в некоторых совершаемых действиях. Например, если конечный узел находится в поддереве *Type* (Набор), его выбор приведет к вводу соответствующего текста в буфер.

Для обозначения узлов с динамически изменяемым заполнением используется префикс ^ . Например, следующий список показывает содержимое файла *type.txt*, который формирует дочерние узлы *Type* в дереве, показанном на рис. 30.1:

```
commonwords.txt
speller
^word completion.txt
^next word.txt
suffixes.txt
^justsaid.txt
^clipboard.txt
^phrase completion.txt
^templates.txt
vocabularytree.txt
```

Поддеревья, чьи имена содержат префикс ^, заполняются, только когда пользователь щелкает на соответствующем корневом узле.

Элемент управления Visual Basic TreeView для ускорения поиска обладает свойством индексации. Это свойство навело нас на мысль о создании узлов в дереве со словами в качестве имен, сгруппированными вместе таким образом, чтобы дочерние узлы в дереве могли заменять друг друга, не создавая абсурдной ситуации. Например, предложение, включающее слово «London», запросто может появиться в другом контексте, имея в этом же месте слово «Boston».

Использование индексации в этом виде позволило нам реализовать два существенных свойства eLocutor, Replace (Замена) и Template (Шаблон), которые будут коротко рассмотрены. Но здесь был и недостаток, который заключался в необходимости мириться с ограничениями свойства индексации элемента управления Tree View, которые не допускали дублирования ключей. Нам ничего не мешало вставить в дерево более одного узла с одним и тем же именем. Но только один из таких узлов мог быть проиндексирован.

Подузел словарного дерева узла Type является корневым узлом большого поддерева, в котором группируются слова, подходящие для взаимной замены в предложении. Чтобы работали Replace (Замена) и Template (Шаблон), они нуждаются в индексации. Тем не менее такое же слово может быть показано и в других местах дерева, возможно, в качестве предложения для завершения текущего слова или в качестве следующего слова. Эти экземпляры не могут быть проиндексированы. Чтобы не усложнять конструкцию, мы решили не индексировать содержимое поддеревьев с динамически обновляемым заполнением.

Speller (Орфографический справочник) обрабатывается особым образом. Содержимое этого узла не является динамическим. Но большое количество содержащихся в нем конечных узлов, не говоря уже о том, что он содержит каждое слово из словарного дерева, означает, что его содержимое также не может быть проиндексировано. Он заполняется по необходимости — то есть дочерние узлы в узле орфографического справочника создаются, только когда он выбран.

Простой набор текста

Поддерево Type (Набор) содержит три узла, способствующие простому набору. В узле Speller (Орфографический справочник) появляются все буквы от а до з, давая возможность выбрать первую букву желаемого слова. Затем вам представляется аналогичный выбор для следующей буквы, но только в том случае, если такая комбинация букв встречается в начале слова, имеющегося в словаре. Таким образом вы выбираете букву за буквой, пока не получится целое слово. В этот момент узел, в котором вы находитесь, может быть, а может и не быть конечным узлом. Если это конечный узел, вы можете ввести его содержимое простым щелчком. Но зачастую этот узел не является конечным.

Рассматриваемые далее «vocabularytree» и «commonwords» являются другими узлами, облегчающими набор текста. Но если системное свойство прогнозирования работает хорошо, что бывает при попытке ввода предложения, подобного одному из хранящихся в базе данных, вам не потребует часто обращаться за помощью к этим средствам.

Прогнозирование: завершение слова и следующее слово

В базе данных прогнозирования имеется несколько таблиц. Одна из них представляет собой простой список, состоящий примерно из 250 000 слов, используемых для заполнения поддерева Word Completion (Завершение слова). Пользователь, набравший один или более начальных символов слова, может воспользоваться этим списком для набора оставшейся части слова, соответствующие предложения показаны в правой части экрана, в его верхней половине (см. рис. 30.1). Эта таблица полностью доступна пользователю в поддереве Speller (Орфографический справочник).

Предположим, вы хотите набрать слово *instant*. Это не конечный узел, поскольку существуют такие слова, как *instantaneous*, которые начинаются с фрагмента *instant*. Поэтому для набора *instant* вы выбираете поочередно каждый из семи символов, и когда будет выделено слово *instant*, используете длинный щелчок для вызова элемента Type This (Набрать это).

Другие таблицы содержат поля *word1*, *word2* и *frequency* (частота повторения). Для заполнения этой таблицы берется длинный список предложений, который предоставляется дополнительной программе dbmanager, а та в свою очередь заносит в таблицу, насколько часто каждое слово следует за каждым другим словом. Как только будет набрано слово, происходит запрос к этой таблице, и поддерево Next Word (Следующее слово) заполняется, предоставляя пользователю список слов, которые могут последовать за этим словом.

Каждое предложение, введенное пользователем с помощью eLocutor, копируется в файл *mailto:mehta@vsnldotcom.txt* (что означает: пошлите мне это по такому-то адресу). Причина выбора такого имени файла заключалась в том, чтобы ненавязчиво предложить пользователю послать мне по почте образец текста, сгенерированного им с помощью программы eLocutor, чтобы я мог получить некоторое представление о том, как сделать работу программы более эффективной. Перед тем как передать этот файл программе dbmanager, пользователю дается совет отредактировать его и удалить все неподходящее, чтобы со временем прогноз улучшился. В том случае если разработчик программного обеспечения хочет реализовать более подходящий метод прогнозирования следующего слова, то ему понадобится всего лишь изменить запрос к базе данных Access; для этого совершенно необязательно ковыряться в коде eLocutor.

В отдельной таблице перечислены сочетания знаков пунктуации, встречающиеся в тексте, поставляемом базе данных, которые в некоторой степени рассматриваются программой eLocutor как слова.

Без знаний семантики программе довольно трудно прогнозировать, что пользователь может захотеть набрать дальше. Мы попытались поговорить с лингвистами, чтобы понять, нет ли каких-нибудь приемлемых и не слишком трудных способов составления подобных прогнозов, но вскоре отказались от этой затеи. Вместо этого мы старательно объединили слова в семантические группы в поддереве Vocabulary tree (Словарное дерево). Например, у названия «Boston» в словарном дереве такая родословная — Nouns ▶ Places ▶ Cities (Существительные ▶ Места ▶ Города). Конечно, пользователь может воспользоваться этим

деревом, чтобы действительно набирать слова, но это не совсем удобно. Семантические подгруппы больше подходят для того, чтобы дать возможность пользователю «заполнить пробелы» в свойствах **Template** (Шаблон) и **Replace** (Замена).

Шаблоны и замена

Пользователь может выбрать любое предложение из базы данных в качестве шаблона для создания новых предложений. Для этого сначала нужно набрать начальное слово или слова, а затем посмотреть в поддерево **Template** (Шаблон). На рис. 30.1 в нижней части экрана присутствуют подсказки по завершению фразы или предложения. Для заполнения этого списка eLocutor обращается к своей базе данных за предложениями, начинающимися с того, что уже было набрано непосредственно после окончания последнего предложения. Такие же подсказки также доступны в поддереве **Template** (Шаблон), с помощью которого пользователь может создать новые предложения путем простого заполнения пробелов в старых образцах. Если подсказка будет слишком много, слово или фраза могут быть помещены в фильтр. Тогда будут показаны только те фразы или предложения, в которых содержится то, что помещено в фильтр.

eLocutor обрабатывает шаблоны, просматривая фразу, выбранную в качестве шаблона, пословно. Любое слово шаблона, не найденное в словарном дереве, выводится в буфер. Для каждого слова, найденного в словарном дереве, eLocutor, используя свойство индексирования TreeView, переносит пользователя в эту часть дерева, позволяя ему выбрать это слово или родственные ему слова. Поэтому если предложение «How are you?» находится в базе данных, пользователю достаточно сделать лишь несколько быстрых щелчков, чтобы было набрано «How is she?». Несмотря на использование такого «заполнения пробелов», та часть шаблона, которая еще не была задействована, отображается в окне **Template** (Шаблон), которое находится под деревом.

Функция **Template** (Шаблон) использует в своих интересах логическую группировку слов в дереве словаря для переделки содержимого всего предложения или фразы. Функция **Replace** (Замена) позволяет пользователю сделать то же самое, но только по отношению кциальному слову, которое было обнаружено последним в среднем окне. Но в словарном дереве представлены не все слова. Поэтому нужно, чтобы в текстовом поле на экране пользователю сообщалось, под какой категорией, если таковая вообще имеется, было найдено рассматриваемое слово. На экране есть поле с надписью **Replace** (Замена). Если последнее слово в буфере найдено в словарном дереве, в текстовом поле **Replace** прописывается имя его родителя.

Например, если последнее слово в буфере – *Boston*, текстовое поле **Replace** содержит слово *Cities* (Города). Тем самым пользователю сообщается, что программа распознала категорию, к которой относится последнее слово. Если затем пользователь выберет команду **Replace** (Замена) (в поддереве **Word Completion** (Завершение слова)), последнее слово удаляется из буфера, и пользователь перемещается в то место словарного дерева, где оно было найдено, что позволяет ему без труда найти на замену другое название города.

На рис. 30.1 последнее введенное слово — We. В поле Replace показано не поместившиеся целиком из-за нехватки места слово subjectpronoun (личное местоимение). Выбор команды Replace приведет к удалению We и перемещению пользователя к поддереву subjectpronoun, в котором он сможет без труда выбрать, к примеру, слово You.

Реализация кэш

Кэширование в программе eLocutor основано на подпрограмме SaveReverse, которая воспринимает два параметра: имя файла, в котором текст должен быть сохранен, и сам текст. Подпрограмма заменяет файл более свежим, в котором текст переданный SaveReverse, является первой строкой файла, за которой следуют первые 19 строк исходного содержимого, не похожего на первую строку.

Это достигается тем, что сначала в качестве первого элемента массива stararray записывается текст, представленный переменной stringtoadd, а остальная часть массива заполняется строками из файла, если только они не являются похожими на stringtoadd (константа, определяющая глубину истории — HistoryLength, имеет значение 20). В завершение файл открывается для записи, что приводит к удалению его предыдущего содержимого, и все содержимое stararray копируется в этот файл.

Таким образом, если название города уже присутствует в перечне файла favouritecities.txt, оно просто изменит свое положение, став первым названием в этом файле. Если будет задействовано новое название города, оно также станет первым в файле, а за ним будут следовать еще 19 строк предыдущего содержимого этого файла. Иными словами, последняя строка файла выбрасывается, и он получает новую первую строку. Как следует из имени подпрограммы, SaveReverse (реверсивное сохранение), строки текста сохраняются в обратном порядке, поэтому последнее использованное слово становится первым.

Код подпрограммы SaveReverse показан в примере 30.2.

Пример 30.2. Добавление текста в начало текстового файла, не содержащего повторений

```
Sub SaveReverse(ByVal filest As String, ByVal stringtoadd As String)
'добавление не к концу, а к началу...
'исключая дубликаты
    Dim stararray(HistoryLength) As String
    Dim i As Long
    Dim arrlength As Long
    Dim st As String
    Dim filenum As Long
    stararray(0) = stringtoadd
    filenum = FreeFile
    i = 1
    On Error GoTo err1
    Open filest For Input As #filenum
    While Not EOF(filenum) And (i < HistoryLength)
        Line Input #filenum, st
        If (st <> stringtoadd) Then 'сохраняются только оригиналы
```

```

        stararray(i) = st
        i = i + 1
    End If
    Wend
arrlength = i - 1
Close #filenum
Open filest For Output As #filenum 'существующее содержимое файла удаляется
For i = 0 To arrlength
    Print #filenum, stararray(i)
Next
Close #filenum
Exit Sub
err1:
    MsgBox "Ошибка обработки файла " + filest
    Open filest For Output As #filenum
    Close #filenum
    Open filest For Input As #filenum 'создание нового файла, если такого файла
    'не существует

Resume Next
End Sub

```

Распространенные и избранные слова

Часто используемые слова собраны в поддереве «common words» (распространенные слова), у которого есть два компонента. Часть этого поддерева неизменна и содержит слова, используемые очень часто, к примеру a, and, but и т. д. Динамически изменяющаяся часть состоит из дополнительных часто используемых пользователем слов, которые находятся в поддереве «favouritechoices» (избранное).

Последние найденные пользователем в орфографическом справочнике Speller 20 слов могут быть обнаружены в его поддереве «favouritespeller» (избранное из справочника). Точно так же если узел, присутствующий в словарном дереве, называется «cities» (города), пользователю нужно лишь создать пустой файл избранных названий городов — favouritecities.txt. После чего последние 20 выбранных пользователем названий, найденных в поддереве городов — «cities», будут доступны в поддереве избранных городов — «favouritecities», поддерева избранное — «favouritechoices». Таким образом, пользователь сам может решать, какую категорию часто используемых слов стоит запомнить и как они могут быть помещены в надлежащее место.

В примере 30.3 показана подпрограмма, создающая новый список избранного и вставляющая его в дерево. Обратите внимание, что stfavorite — это константа избранного, а MakeFullFileName возвращает подходящее имя файла, включая путь, имя и расширение .txt.

Пример 30.3. Как eLocutor помещает уже набранные слова в избранное

```

Public Sub AddToFavourites(parenthead As Node, stAdd As String)
Dim tempfilename As String
If parenthead.Text = stStart Then

```

```

    Exit Sub
End If
tempfilename = MakeFullFileName(App.Path, stfavourite + parentnode.Text)
If FileExists(tempfilename) Then
    SaveReverse tempfilename, stAdd
Else
    AddToFavourites parentnode.Parent, stAdd
End If
End Sub

```

Когда слово набрано, eLocutor рассчитывает, что оно также может быть найдено в словарном дереве. Предположим, что только что было набрано слово Boston. В этом случае Boston вставляется в начало файла избранных городов – favouritecities.txt, если таковой существует, для чего используется подпрограмма SaveReverse. Если он не существует, eLocutor ищет файл избранных мест – favouriteplaces.txt, поскольку родительским узлом для городов – Cities служит узел мест – Places. Если и этот файл не существует, eLocutor пытается обратиться к более отдаленному предку. Если существует файл избранных мест – favouriteplaces.txt, Boston добавляется к этому файлу, для чего используется та же подпрограмма. Таким образом, пользователь получает управление тем, что программа должна рассматривать в качестве его «избранного». Путем создания файла под названием favouritecities.txt, он сообщает программе eLocutor, что он часто пользуется названиями городов.

Отслеживание путей

В целях быстрого перемещения в слишком большом дереве eLocutor автоматически запоминает для каждого поддерева, в котором пользователь выбирал элементы, что этот пользователь делал последние 20 раз после того, как он сделал в нем свой выбор. Эти информационные адресаты представляются в удобном для пользователя виде. Каждый родительский узел x имеет поддерево x_Next (следующий x). После выбора конечного узла пользователь может заглянуть в родственный _Next-узел и выбрать адресат информации, близкий к тому, на который он хочет далее перейти. eLocutor довольно эффективно обнаруживает примеры действий, предпринимаемых пользователем, позволяя ему с легкостью их повторять. Программа также запоминает последние 20 открывавшихся файлов, последние 20 образцов разыскивавшегося текста и последние 20 предложений, упомянутых пользователем. Все это легко реализуется с помощью подпрограммы SaveReverse.

Буфер набора, редактирование и прокрутка

Управлять прокруткой текста, его выбором для вырезки и вставки можно несколькими различными способами. Большинство редакторов работают с одним окном. Конечно, при работе с большим документом весь текст не помещается в отображаемом окне, и для перехода по тексту используются полосы прокрут-

ки. Когда нужно скопировать или вырезать текст, его нужно сначала выбрать. Выбранный текст выделяется с использованием различных цветов шрифта и фона. Но у нас с таким стандартным подходом возникла ряд проблем.

Нам хотелось, чтобы eLocutor мог использоваться лицами, страдающими церебральным параличом, часто испытывающими серьезные двигательные расстройства, выражющиеся в ухудшении речевых и зрительных функций. Для них нам нужно было отображать хотя бы часть текста очень крупным шрифтом. Если использовать это свойство для всего текста, то объем того, что помещается на экране, будет крайне незначителен. Мы считали, что использование слишком крупного шрифта для части текста в окне будет сопряжено с некоторыми неудобствами. Текст, выделенный для вырезки и вставки путем изменения фонового цвета, некоторые считали отвлекающим внимание и затрудняющим чтение. Исходя из нашего опыта и пристрастия к звукомонтажу мы пришли к выбору иной парадигмы.

В прежние времена, когда звукомонтаж делался на бобинах с магнитной лентой, режиссер должен был прослушивать содержимое ленты до тех пор, пока не находил начало той части, которую хотел вырезать, устанавливал фиксатор, затем прослушивал ленту до конца удаляемого фрагмента и сноваставил фиксатор. Теперь участок между фиксаторами можно было вырезать или заменить чем-нибудь другим. Таким образом лента делилась двумя фиксаторами на три отрезка: на тот, что был до фиксатора № 1, на тот, что был после фиксатора № 2, и на тот отрезок, который был между фиксаторами.

Мы применили такой же подход к тексту, поделив его на три текстовых окна с воротами между ними. Весь набор осуществляется в конце текста в среднем окне. Именно в нем происходит реальная вставка и удаление текста. Элемент Backspace (Забой), входящий в Edit (Редактирование), удаляет текст в среднем окне с конца. Вы можете решить, хотите ли вы избавиться от символа, слова, фразы, предложения, абзаца или всего содержимого среднего окна.

Если вы выберете в Edit (Редактирование) Cut (Вырезать) или Copy (Копировать), весь текст в среднем окне будет скопирован в буфер обмена. Разумеется, Cut оставит среднее окно пустым. Чтобы сравнить это с обычными редакторами, позволяющими устанавливать начало и конец текстового блока, который вы собираетесь вырезать или копировать, представьте, что блок начинается на границе между верхним и средним окном и заканчивается на границе между средним и нижним окном. Команда Cut (Вырезать) или Copy (Копировать) всегда забирает все содержимое среднего окна.

Такое распределение текста по нескольким окнам позволяет нам более рационально использовать пространство экрана. Текст в верхнем окне мы показываем только в процессе прокрутки. В других режимах мы можем использовать его для отображения выделенного элемента дерева крупным шрифтом, как показано на рис. 30.1, или содержимого низших уровней дерева, предоставляемого пользователю возможность «заглянуть вперед». Точно так же в определенные моменты мы используем пространство нижнего окна для отображения меню длинного щелчка.

При определении наилучшего варианта распределения пространства экрана мы действовали методом проб и ошибок. Когда отдельные пользователи высказы-

зывали свои особые пожелания о внешнем виде экрана, мы старались приспособить для осуществления их запросов верхнее и нижнее окно.

Аналогом фиксаторов, используемых при звукомонтаже, у нас служат ворота. Если нужно вырезать большой фрагмент текста, то сначала нужно осуществить прокрутку, пока начало этого фрагмента не окажется в начале среднего окна. Затем мы закрываем в верхнем окне ворота между окнами, и прокрутка уже не приводит к перемещению по тексту, находящемуся за пределами этой границы: текст в данном месте «фиксируется». Затем мы продолжаем прокрутку вниз или вверх до тех пор, пока окончание того фрагмента, который нужно вырезать, не появится в конце среднего окна. Теперь в команде Edit (Редактирование) можно выбрать подкоманду Cut (Вырезать).

Элементы меню, входящие в Scroll (Прокрутка), позволяют воротам открыться поодиночке или обоим сразу. Состояние ворот отображается красными или зелеными окружностями. На рис. 30.1 все ворота открыты, о чем свидетельствуют зеленые окружности справа и слева от среднего окна. Для перемещения текста между окнами доступны две команды: Text Up (Текст вверх) и Text Down (Текст вниз). Чтобы текст мог перемещаться между верхним и средним окном, или между средним и нижним окном, должны быть открыты соответствующие ворота.

Объем текста, перемещаемого по командам вверх и вниз, зависит от выбранного пользователем маркера, который может быть символом, словом, знаком препинания, предложением или абзацем. Текущий выбранный маркер прокрутки отображается на экране под деревом. Команды также позволяют перемещать все содержимое текстовых окон из одного окна в другое.

Чтобы иметь возможность прокрутки небольшого объема текста в процессе его набора, можно применить длинный щелчок и получить доступ к командам Words Down (Слова вниз) и Words Up (Слова вверх). Когда выбрана одна из этих команд, слова прокручиваются в избранном направлении, пока не будет сделан еще один щелчок правой кнопкой мыши. Учтите, что сочетания знаков пунктуации также рассматриваются в качестве слов. Это дает возможность пользователю вносить немедленные правки в местах вставки или удаления текста, осуществляя быструю прокрутку в процессе набора.

Буфер обмена

Когда пользователь в поддереве Edit (Редактирование) выбирает команду Cut (Вырезать) или Copy (Копировать), вызывается подпрограмма SaveReverse, чтобы присоединить содержимое среднего окна в начало файла clipboard.txt, в котором содержится в общей сложности 20 абзацев. Преимущества такого подхода состоят в том, что он позволяет абзацам легко перестраиваться, а последнему фрагменту быть вставленным несколько раз. Во многих текстовых редакторах при каждом выборе команды Cut (Вырезать) или Copy (Копировать) предыдущее содержимое буфера обмена утрачивается. В eLocutor некоторое время доступна прежняя информация, содержащаяся в буфере обмена.

ПОИСК

Ни один уважающий себя редактор не может обойтись без функции поиска, но eLocutor дает нам возможность взглянуть на эту основную функцию с новой стороны. Мы поняли, что поиск — это всего лишь особый случай прокрутки, поэтому мы просто расширили реализацию этой функции. Пользователь может скопировать текст из среднего окна в буфер фильтра или воспользоваться в отношении выделенного в дереве желаемого текста командой длинного щелчка **Set Filter** (Установить фильтр). Когда в фильтре есть текст и задана команда прокрутки, то прокрутка не останавливается до тех пор, пока содержимое фильтра не будет найдено в среднем окне или не будет достигнут конец текста.

Макрокоманда

Во время обсуждения в кабинете профессора Хокинга возник довольно интересный момент. Мне сказали, что иногда у него возникают проблемы с системой Equalizer при произнесении речи, когда освещение затрудняет ему чтение с экрана. Не имея возможности читать с экрана, он испытывал затруднения в произвольной прокрутке текста, когда запускал команду озвучки.

В eLocutor уже была возможность помещать весь текст речи в среднее окно и выдавать программе команду на его озвучку, но этого было недостаточно. Люди в процессе речи могли аплодировать или смеяться, поэтому нужна была возможность подождать, пока они успокоятся, перед тем как продолжить чтение лекции.

Для нас не представляло труда встроить функцию для прокрутки и озвучки предложения, когда пользователь выберет соответствующий пункт меню, но вместо того чтобы жестко задать эту функцию, мы решили, что будет лучше перенести решение этой задачи на более общий уровень, предоставив функцию создания макрокоманд, позволяющую в будущем создавать и другие подобные комбинации.

В поддереве **Commands** (Команды) имеется узел под названием **Macros** (Макрос), в котором приводится список всех файлов из подкаталога **C:\eLocutor\macros**. Выбор любого из этих файлов приводит к его открытию и поочередному выполнению перечисленных в нем команд. В конструкции макрокоманды отсутствуют какие-либо сложные элементы, в ней нет ни переходов, ни циклов, ни ветвлений.

Для произнесения речи мы создали две небольшие макрокоманды, **rgerarespeech** и **scrollspeak**. Макрокоманда **rgerarespeech** открывала все ворота, если они еще не были открыты, и помещала весь текст в нижнее окно. После того как макрос выполнялся, пользователь выполнял команду длинного щелчка **Pause** (Пауза), когда была выбрана макрокоманда **scrollspeak**. Все это могло быть сделано заранее.

Организовав все это, пользователь мог уже не смотреть на экран. С каждым новым щелчком запускалась макрокоманда **scrollclick**, фактически выполнявшая две команды. Первая команда — это **Text Up** (Текст вверх), которая посыпала столько текста, сколько было указано в маркере прокрутки из нижнего окна

в среднее и из среднего окна в верхнее. Вторая команда озвучивала содержимое среднего окна. Обычно для произнесения речи маркер прокрутки устанавливался на предложение, поэтому речь произносилась по предложениям, но для достижения большей гибкости он мог устанавливаться и на абзац.

Эффективность пользовательского интерфейса

Чтобы определить эффективность помощи eLocutor в наборе текста, в правом нижнем углу экрана выведены два числа (рис. 30.2). Они показывают количество щелчков и количество секунд между последним и первым щелчком, с тех пор как средний экран был в последний раз опустошен.

Мы поняли, что когда прогнозирование работает достаточно хорошо, соотношение щелчков к набранным символам было лучше, чем 0,8 – то есть обычно требовалось значительно меньше щелчков, чем потребовалось бы здоровому человеку, вынужденному пользоваться полноценной клавиатурой. Когда прогнозирование было не на высоте – например, когда составлялось предложение, в корне отличающееся от находящихся в базе данных, – требовалось чуть ли не вдвое больше щелчков по отношению к количеству вводимых символов.

Загрузка

eLocutor – это свободная программа с открытым кодом, доступная для загрузки по адресу <http://holisticit.com/eLocutor/elocutorv3.htm>. Ее обсуждение ведется на веб-сайте <http://groups.yahoo.com/group/radiophony>.

Загружаемая часть представляет собой весь исходный текст. Я вправе вас предупредить, что он отчасти похож на сплетение макарон, за которое я несу полную ответственность. Когда я начинал работу над проектом, я уже не программирую более 10 лет, поэтому мои навыки устарели и немного «заржавели». У меня не было цельного замысла, были лишь некоторые наметки и направление, в котором они должны были развиваться. Код рос в объеме по мере осмысливания проблемы, что и отразилось на результатах. Была использована очень простая технология программирования, что становится понятным из кода, представленного в этой главе.

Будущие направления

eLocutor всегда рассматривался в качестве проекта, связанного с быстрой разработкой приложений – rapid application development (RAD)¹, как средство демонстрации профессору Хокингу во время наших нечастых и кратковременных встреч, насколько далеко вперед ушла конструкторская мысль. Было намерение со временем переписать эту программу, когда конструкция могла быть зафик-

¹ http://en.wikipedia.org/wiki/Rapid_application_development.

сирована на уровне работающего прототипа, который уже мог использоваться людьми и позволял получать от них отзывы. К тому моменту я бы подобрал кроссплатформенный язык программирования, чтобы системы, использующие Mac или Linux, также были доступны для людей с расстройствами движений.

Но вдохновленный тем, что Т. В. Раман (T. V. Raman) достиг в разработке программы Emacspeak (рассматриваемой в главе 31), теперь я обдумываю совершенно новую версию проекта. Разумеется, что сам Emacs – это не только редактор, а довольно-таки универсальная платформа, которая с годами была расширена и теперь дает возможность чтения электронной почты, управления оборудованием, просмотра Интернета, исполнения команд оболочки и т. д. Путем простого добавления интеллектуального текстово-речевого преобразования и контекстно-зависимых команд Раман блестяще организовал доступ незрячим людям ко всему, что может быть получено с использованием Emacs.

Поэтому я заинтересовался, а нельзя ли сделать то же самое для людей с расстройствами движений. У этого подхода есть следующие преимущества.

- Теперь конструкторы могут отказаться от мыши, поскольку Emacs позволяет все делать и без нее.
- eLocutor стал бы не только доступным редактором, но, скорее всего, сделал бы доступными все возможности, предоставляемые компьютером.
- Работая в этом направлении, я смог бы также получить больше поддержки от сообщества разработчиков программ с открытым кодом, которая представляется намного более продуктивной на платформах, исторически тесно связанных с использованием Emacs, чем та, которую я смог бы получить в мире MS Windows.

Поэтому я обращаюсь к читателям этой главы, чтобы они научили меня, как можно расширить Emacs, чтобы в нем стали доступны такие же однокнопочные перемещения по дереву. Еще лучше было бы, если кто-нибудь захотел взяться за этот проект, рассчитывая на мою всестороннюю помощь.

Другим направлением использования этой программы могло стать обращение к очень серьезной проблеме тех детей, кто стал инвалидом в первые годы своей жизни, тех, кто страдает церебральным параличом и тяжелой формой аутизма, кто обычно не получает образования, поскольку не может отвечать на вопросы учителя в обыкновенном школьном классе. Если такие дети смогут общаться при помощи программного обеспечения, они смогут посещать обычную школу.

Здесь сложность задачи, возлагаемой на программиста, значительно возрастает. Обычно мы предполагаем, что человек, использующий компьютер, владеет грамотой. В данном случае дети должны иметь возможность использовать компьютер, чтобы *стать* грамотными. Создаваемое нами программное обеспечение должно быть настолько привлекательным для ребенка, чтобы он испытывал желание использовать его в качестве основного средства общения с окружающим миром еще до того, как научится читать. Несмотря на свою отступающую сложность, это все же весьма интересная задача! Разумеется, программное обеспечение пригодилось бы не только детям-инвалидам, но прекрасно подошло бы и для обучения любого ребенка использованию компьютера в самом раннем возрасте. Ну что, кто-нибудь согласен сотрудничать?

31 Emacspeak: полноценно озвученный рабочий стол

T. Раман (T. V. Raman)

Рабочий стол является пространством, используемым для раскладки рабочего инструмента. Графические рабочие столы обеспечивают великолепное визуальное взаимодействие для выполнения ежедневных задач, связанных с обработкой данных; предназначение звукового рабочего стола состоит в том, чтобы дать возможность осуществлять такие же полезные действия в невизуальной среде. Таким образом, главная задача звукового рабочего стола состоит в использовании выразительных возможностей звуковой выходной информации (как в словесной, так и в простой звуковой форме) для того, чтобы позволить конечному пользователю выполнять полный спектр задач по обработке данных.

- связь с использованием всевозможных электронных служб сообщений;
- свободный доступ к локальным документам на стороне клиента и к глобальным документам Интернета;
- возможность эффективной разработки программного обеспечения в невизуальной среде.

Создание звукового рабочего стола Emacspeak было вызвано пониманием следующего положения: для обеспечения эффективной звуковой интерпретации нужно начинать с самой сути представляемой информации, а не с ее визуального представления. Ранее это понимание привело меня к разработке аудиосистемы для технического чтения – AsTeR, Audio System For Technical Readings (<http://emacspeak.sf.net/raman/aster/aster-toplevel.html>). Тогда основной мотивацией было применение уроков, извлеченных из создания звуковых документов, к пользовательским интерфейсам – в конце концов, документ *и есть* интерфейс.

Первичная цель заключалась не в простом переносе визуального интерфейса в область применения аудиосредств, а в создании невизуального пользовательского интерфейса, удобного и эффективного в использовании.

Нужно было что-то противопоставить традиционному подходу считывания с экрана, где элементы графического пользовательского интерфейса, такие как

ползунки прокрутки и элементы управления деревом, напрямую преобразовывались в речевой вывод. Хотя подобный прямой перевод мог бы создать иллюзию полного невизуального доступа, получающийся при этом звуковой пользовательский интерфейс мог быть неэффективным в использовании.

Эти предварительные условия означали, что среда, выбираемая для звукового рабочего стола, нуждалась в следующем:

- в основном наборе речевых и неречевых служб звукового вывода;
- в богатом наборе готовых говорящих приложений;
- в доступе к содержимому приложения для осуществления контекстной обратной связи.

Создание речевого вывода

Я приступил к реализации Emacspeak в октябре 1994 года. Исполнительной средой служил переносной компьютер под управление Linux и рабочая станция, находящаяся в моем офисе. Для создания речевого вывода я воспользовался DECTalk Express (аппаратным речевым синтезатором) на переносном компьютере и программной версией DECTalk на офисной рабочей станции.

Самый естественный способ проектирования системы для расширения возможностей обоих речевых средств заключался в предварительной реализации речевого сервера, позволяющего абстрагироваться от различий между двумя решениями речевого вывода. Абстракция в виде речевого сервера неплохо выдержала испытания временем; позже, в 1999 году, у меня появилась возможность добавить поддержку движка IBM ViaVoice. Кроме того, простота клиент-серверного API дала возможность программистам, работающим с открытым кодом, реализовать речевые серверы для других речевых движков.

Речевые серверы Emacspeak реализованы на языке TCL. Речевой сервер для DECTalk Express обменивался данными с аппаратным синтезатором по последовательному каналу. К примеру, команда на произнесение строки текста называлась proc, воспринимала строковый аргумент и записывала его в последовательное устройство. Ее упрощенная версия имела следующий вид:

```
proc tts_say {text} {puts -nonewline $tts(write) "$text"}
```

Речевой сервер для программного DECTalk приводил в исполнение эквивалентную, упрощенную версию tts_say, которая выглядела следующим образом:
proc say {text} {_say "\$text"}.

где _say вызывает базовую Си-реализацию, предоставляемую программным обеспечением DECTalk.

Полезный результат от такой конструкции состоял в создании отдельных речевых серверов для каждого доступного движка, где каждый речевой сервер был простым сценарием, который после загрузки с соответствующими определениями вызывал исходный TCL-цикл считывания-обработки-вывода. Поэтому клиент-серверный API добирался до клиента (Emacspeak), запускал соответствующий речевой сервер, кэшировал это подключение и вызывал команды сервера, выдавая соответствующие вызовы процедур через это подключение.

Заметьте, что я до сих пор еще не сказал ничего конкретного о том, как было открыто это клиент-серверное подключение; это более поздняя привязка, которая стала полезной несколько позже, когда настало время подготовить Emacspeak к сетевой работе. Но начальная реализация, работавшая с клиентом Emacspeak, связывалась с речевым сервером с использованием `stdio`. Позже осуществление этой клиент-серверной связи по сети потребовало добавления нескольких строк кода, которые открывали серверный сокет и связывали `stdin/stdout` с получившимся подключением.

Таким образом, разработка вполне понятной клиент-серверной абстракции и надежда на мощь, присущую системе ввода-вывода Unix, облегчили последующий запуск Emacspeak на удаленном компьютере и его обратное подключение к речевому серверу, запущенному на машине клиента. Это позволило мне запустить Emacspeak за экраном своего рабочего компьютера и обратиться к этой работающей сессии из любой точки мира. Через соединение я получаю сессию удаленного подключения Emacspeak к речевому серверу на моем переносном компьютере, звуковой эквивалент установки X для использования удаленного дисплея.

Говорящий Emacs

Описанная выше простота абстракции речевого сервера означала что версия 0 речевого сервера заработала через час после начала реализации системы. А это значит, что я мог перейти к более интересной части проекта: созданию высококачественного речевого вывода. Версия 0 речевого сервера отнюдь не отличалась совершенством; она была реализована на время создания речевого клиента Emacspeak.

Простая пробная реализация

За несколько недель до этого мой друг обратил мое внимание на чудеса, на которые способно имеющееся в Emacs Lisp средство `advice`. И когда я засел за говорящий Emacs, `advice` стал для меня вполне естественным выбором. Начальной задачей было заставить Emacs автоматически произносить строку под курсором, когда пользователь нажимал клавиши стрелок вверх и вниз.

В Emacs все действия пользователя вызывают соответствующие функции Emacs Lisp. В стандартных режимах редактирования нажатие стрелки вниз вызывает функцию `next-line`, а стрелки вверх — функцию `previous-line`. Чтобы сделать эти команды говорящими, в версии 0 Emacspeak был реализован следующий довольно простой `advice`-фрагмент:

```
(defadvice next-line (after emacspeak)
  "Озвучивание строки после перемещения."
  (when (interactive-p) (emacspeak-speak-line)))
```

Функция `emacspeak-speak-line` обеспечивала выполнение логических построений, необходимых для захвата текста строки, находящейся под курсором, и отправки ее на речевой сервер. Именно с этого, предыдущего определения и взял

свое начало Emacspeak 0.0; он предоставил временную платформу для построения реально действующей системы.

Подходы к пробной реализации

Следующим шагом я вернулся к речевому серверу, чтобы усилить его четким циклом обработки событий. Вместо простого выполнения каждой полученной им речевой команды, речевой сервер выстраивал клиентские запросы в очередь и выдавал команду `launch`, которая заставляла сервер выполнять выстроенные в очередь запросы.

Для проверки вновь поступивших команд после отправки каждого оператора речевому движку сервер использовал системный вызов `select`. Это позволяло немедленно прекращать речь; в сравнительно простой реализации, описанной в версии 0 речевого сервера, команда на остановку речи не приводила к немедленному результату, поскольку речевой сервер сначала должен был завершить прежде выданные команды `speak`. Благодаря наличию речевой очереди клиентское приложение теперь могло выставлять в очередь произвольное количество текста и сохранять высокую степень реагирования при выдаче команд с высоким приоритетом, наподобие запросов на остановку речи.

Кроме этого, реализация в речевом сервере очереди событий позволила клиентскому приложению осуществлять детальный контроль над тем, как текст был разбит на части перед синтезом. Оказалось, что это весьма существенная деталь для выработки хорошей интонационной структуры. Правила, по которым должно происходить разбиение текста на элементарные части, меняются в зависимости от сути произносимого текста. К примеру, символы разделителей строк в таком языке программирования, как Python, являются разделителями операторов и определяют границы фраз, но в тексте английского языка разделители строк не предназначены для разделения фраз.

К примеру, если речь идет о следующем коде, написанном на Python, то граница фразы вставляется после каждой строки:

```
i=1  
j=2
```

Подробности распознавания кода, написанного на языке Python, и переноса его семантики на речевой уровень описаны далее в этой главе в разделе «Дополнение Emacs для создания таблиц звукового отображения».

Теперь, когда речевой сервер стал способен к интеллектуальной обработке текста, клиент Emacspeak также стал более изощренным по отношению к обрабатываемому им тексту. Функция `emacspeak-speak-line` превратилась в библиотеку функций генерации речи, осуществляющую следующие действия.

1. Разбор текста для разбиения его на последовательность фраз.
2. Предобработку текста — например обработку повторяющихся последовательностей знаков препинания.
3. Выполнение многих других функций, добавленных впоследствии.
4. Постановку каждой фразы в очередь речевого сервера и выдачу команды запуска.

Начиная с этого момента все остальное в Emacspeak было реализовано с использованием Emacspeak в качестве среды разработки. Это сыграло значительную роль в развитии самой основы программы. Новые функции проходили немедленную проверку, поскольку некачественно реализованные функции могли сделать неработоспособной всю систему. Пошаговая разработка кода на Lisp хорошо вписывалась во все ранее созданное; чтобы включить в себя последнюю наработку, исходный код Emacspeak развивался так, чтобы стать более «разветвленным» — например, многие части высокоуровневой системы не зависят друг от друга, они зависят лишь от небольшого ядра, которое прошло очень тщательную отработку.

Краткое руководство по advice

Ключом к реализации Emacspeak является имеющееся в Lisp средство *advice*, и эта глава не может обойтись без его краткого обзора. Средство *advice* позволяет модифицировать существующие функции *без изменения их оригинальной реализации*. Более того, если функция *f* была модифицирована с помощью *advice*-средства *t*, все вызовы функции *f* окажутся под влиянием *advice*.

Средство *advice* имеет три разновидности.

before

Тело *advice* запускается *перед* вызовом оригинальной функции.

after

Тело *advice* запускается *после* завершения работы оригинальной функции.

around

Тело *advice* запускается *вместо* оригинальной функции. Если нужно, то *around* может вызвать оригинальную функцию.

Все формы *advice* получают доступ к аргументам *обрабатываемой* этим средством функции; вдобавок к этому *around* и *after* получают доступ к возвращаемому значению, вычисленному оригинальной функцией. Реализация Lisp достигает этого волшебства за счет:

- 1) кэширования оригинальной реализации функции;
- 2) определения разновидности *advice* для генерирования нового определения функции;
- 3) сохранения этого определения в качестве *обработанной* средством *advice* функции.

Поэтому когда происходит вычисление фрагмента *advice*, показанного в предыдущем разделе «Простая пробная реализация», оригинальная функция Emacs *next-line* подменяется модифицированной версией, которая озвучивает текущую строку *после* завершения работы оригинальной функции *next-line*.

Генерация полноценного речевого вывода

На данном этапе развития общая конструкция выглядела следующим образом.

1. Интерактивные команды Emacs стали говорящими или *обработанными advice* таким образом, чтобы выдавать на выходе речь.

2. Определения *advice* собраны в модули, по одному для каждого Emacs приложения, получившего речевой вывод.
3. Формы *advice* отправляют текст к основным речевым функциям.
4. Эти функции извлекают произносимый текст и отправляет его функции *tts-speak*.
5. Функция *tts-speak* вырабатывает речевой вывод, предварительно обрабатывая свой аргумент *text*, и отправляет его на речевой сервер.
6. Речевой сервер обрабатывает выстроенные в очередь запросы для выработки воспринимаемого вывода.

Предварительная обработка текста осуществляется за счет помещения его в специальный рабочий буфер. Буфера получают специализированное поведение посредством характерных для каждого буфера *синтаксических таблиц*, которые определяют *грамматику* содержимого буфера и локальные переменные буфера, влияющие на его поведение. Когда текст передан ядру *Emacspeak*, все эти характерные для буфера настройки распространяются на специальный рабочий буфер, где текст проходит предварительную обработку. Тем самым автоматически гарантируется, что текст разбивается по смыслу на фразы относительно своей основной грамматики.

Аудиоформатирование с использованием *voice-lock*

Emacs использует *font-lock*, чтобы придать тексту синтаксическую окраску. Для создания визуального представления Emacs добавляет к тестовой строке текстовое свойство под названием *face*; значение этого свойства *face* определяет шрифт, цвет и стиль, используемые для отображения этого текста. Текстовые строки, имеющие свойство *face*, могут рассматриваться в качестве *таблиц визуального отображения*. *Emacspeak* наращивает эти таблицы визуального отображения за счет текстового свойства *personality*, чьи значения определяют звуковые свойства, используемые при интерпретации заданного текстового фрагмента; в *Emacspeak* это называется *voice-lock*. Значение свойства *personality* – это звуковая CSS (Aural CSS – ACSS), установка, которая кодирует различные голосовые свойства, например высоту голоса. Заметьте, что такие параметры настройки ACSS не являются специфическими для любого заданного речевого движка (TTS engine). *Emacspeak* реализует ACSS-to-TTS отображение в специфических для движков модулях, которые берут на себя отображение высокоуровневых свойств звучания, например отображение высоты голоса – *pitch* или *pitch-range*, на специфические для движка управляющие коды.

В следующих нескольких разделах будет рассмотрено, как *Emacspeak* дополняет Emacs для создания таблиц звукового отображения и для обработки этих таблиц в целях создания специфического для данного движка выхода.

Дополнение Emacs для создания таблиц звукового отображения

Модули Emacs, реализующие font-lock, для прикрепления значимого свойства face вызывают встроенную в Emacs функцию put-textproperty. Emacspeak определяет *advice*-фрагмент, который обрабатывает функцию put-text-property в целях добавления соответствующего свойства personality, когда она запрашивается для добавления свойства face. Заметьте, что значения обоих свойств отображения (face и personality) могут быть таблицами; таким образом, значения этих свойств сконструированы с расчетом на *каскадную запись* для создания окончательного (визуального или звукового) представления. Это также означает, что различные части приложения могут постепенно добавлять значения свойств отображения.

Функция put-text-property имеет следующие характерные признаки:

(put-text-property START END PROPERTY VALUE &optional OBJECT)

Реализация *advice* имеет следующий вид:

```
(defadvice put-text-property (after emacspeak-personality pre act)
  "Используется emacspeak для дополнения font lock."
  (let ((start (ad-get-arg 0)) :: Привязка аргументов
        (end (ad-get-arg 1)))
    (prop (ad-get-arg 2)) :: имя добавляемого свойства
    (value (ad-get-arg 3))
    (object (ad-get-arg 4))
    (voice nil)) :: голос, на который производится отображение
  (when (and (eq prop 'face)) :: избавление от бесконечной рекурсии
    (not (= start end)) :: ненулевое текстовое пространство
    emacspeak-personality-voiceify-faces)
  (condition-case nil :: безопасный поиск face-отображения
    (progn
      (cond
        ((symbolp value)
         (setq voice (voice-setup-get-voice-for-face value)))
        ((ems-plain-cons-p value)) :: передача простых условий
        ( (listp value)
          (setq voice
                (delq nil
                      (mapcar #'voice-setup-get-voice-for-face value))))
        (t (message "Got %s" value)))
      (when voice :: voice содержит таблицу personalities
        (funcall emacspeak-personality-voiceify-faces start end voice object)))
    (error nil))))
```

Вот краткое объяснение этого *advice*-определения.

Привязка аргументов

Сначала функция использует встроенную в *advice* функцию ad-get-arg для локальной привязки набора лексических переменных к аргументам, переданным обрабатываемой *advice* функции.

Установщик свойства personality

Отображение свойств face на свойства personality управляется настраиваемой пользователем переменной emacspeak-personality-voiceify-faces. Если

значение ненулевое, эта переменная определяет функцию со следующими характерными признаками:

(emacspeak-personality-put START END PERSONALITY OBJECT)

Emacspeak предоставляет различные реализации этой функции, которые либо добавляют новое значение свойства *personality* в конец, либо ставят его впереди любых уже существующих свойств *personality*.

Защита

Наряду с проверкой на ненулевое значение *emacspeak-personality-voiceify-faces* функция осуществляет ряд дополнительных проверок, чтобы выяснить, должно ли данное *advice*-определение что-нибудь делать. Функция продолжает действовать, если:

- текстовое пространство не равно нулю;
- добавленное свойство является свойством *face*.

Первая из этих проверок предназначена для исключения крайних случаев, когда *put-text-property* вызывается с текстом нулевой длины. Вторая проверка гарантирует, что мы пытаемся добавить свойство *personality*, только когда свойство, к которому оно добавляется, является свойством *face*. Заметьте, что отсутствие этой второй проверки приведет к бесконечной рекурсии, поскольку случайный вызов *put-textproperty*, который добавляет свойство *personality*, также вызывает *advice*-определение.

Получение отображения

После этого функция *безопасно* проводит поиск голосового отображения примененного свойства *face* (или нескольких таких свойств). Если применяется отдельное свойство *face*, функция ищет соответствующее отображение свойства *personality*; если используется таблица свойств *face*, она создает соответствующую таблицу свойств *personality*.

Применение свойства *personality*

И наконец, функция проверяет, что ею найдено правильное голосовое отображение, и в случае успешной проверки вызывает *emacspeak-personality-voiceify-faces* с набором свойств *personality*, сохраненных в переменной *voice*.

Выход, имеющий звуковое форматирование на основе таблиц звукового отображения

Теперь, благодаря *advice*-определениям из предыдущего раздела, текстовые фрагменты, имеющие визуальное стилевое оформление, получили соответствующее свойство *personality*, которое содержит установки ACSS для звукового форматирования содержимого. Результат выражается в превращении текста в Emacs в полноценные звуковые таблицы отображения. В этом разделе описывается, как выходной уровень *Emacspeak* расширен для преобразования этих звуковых таблиц отображения в понятный речевой вывод.

Модуль Emacspeak tts-speak занимается предварительной обработкой текста перед окончательной его отправкой на речевой сервер. Как уже ранее описывалось, эта предварительная обработка охватывает несколько этапов, включая:

- 1) применение правил произношения;
- 2) обработку повторяющихся последовательностей символов пунктуации;
- 3) разбиение текста на соответствующие фразы, основанные на контексте;
- 4) преобразование свойства `personality` в коды звукового форматирования.

В этом разделе описывается функция `tts-format-text-and-speak`, которая управляет преобразованием звуковых таблиц отображения в выход, имеющий звуковое форматирование. Сначала рассмотрим код функции `tts-format-text-and-speak`:

```
(defsubst tts-format-text-and-speak (start end )
  "Форматирование и озвучивание текста между start и end."
  (when (and emacspeak-use-auditory-icons
             (get-text-property start 'auditory-icon)) ;: обозначение очереди
    (emacspeak-queue-auditory-icon (get-text-property start 'auditory-icon)))
  (tts-interp-queue (format "%s\n" tts-voice-reset-code))
  (cond
    (voice-lock-mode :: звуковое форматирование только при включенном
                      :: режиме voice-lock-mode)
    (let ((last nil) :: инициализация
          (personality (get-text-property start 'personality)))
      (while (and (
        < start end ) :: фрагмент в изменениях personality
        (setq last
              (next-single-property-change start 'personality
                (current-buffer) end)))
        (if personality :: фрагмент звукового формата
            (tts-speak-using-voice personality (buffer-substring start last ))
            (tts-interp-queue (buffer-substring start last)))
        (setq start last :: подготовка к следующему фрагменту
              personality (get-text-property last 'personality))))
      :: если нет voice-lock, отправка одного текста
      (t (tts-interp-queue (buffer-substring start end )))))
    ))
```

Функция `tts-format-text-and-speak` вызывается для каждой отдельной фразы с аргументами `start` и `end`, устанавливающими начало и конец фразы. Если режим `voice-lock-mode` включен, эта функция продолжает разбиение фразы на фрагменты там, где у текста изменяется значение свойства `personality`. Как только такое место перехода будет обнаружено, `tts-format-text-and-speak` вызывает функцию `tts-speak-using-voice`, передавая в ее распоряжение свойство `personality` и произносимый текст. Эта функция, описываемая далее, проводит поиск соответствующих, характерных для данного устройства кодов, перед тем как выдать отформатированный звуковой сигнал речевому серверу:

```
(defsubst tts-speak-using-voice (voice text)
  "Использование голоса VOICE для произношения текста TEXT."
  (unless (or (eq 'inaudible voice) :: не произносить, если голос не слышен
              (and (listp voice) (member 'inaudible voice)))
    (tts-interp-queue
      (format "%s%s %s \n"
      (cond
```

```
((symbolp voice)
  (tts-get-voice-command
    (if (boundp voice) (symbol-value voice) voice)))
((listp voice)
  (mapconcat #'(lambda (v)
    (tts-get-voice-command
      (if (boundp v) (symbol-value v) v)))
  voice
  " "))
(t ""))
  text tts-voice-reset-code))))
```

Функция `tts-speak-using-voice` осуществляет немедленный возврат управления, если указанный голос не слышим — `inaudible`. Здесь `inaudible` — специальное значение свойства `personality`, используемое `Emacspeak` для того, чтобы не озвучивать какие-то части текста. Значение `inaudible` свойства `personality` может быть использовано, чтобы выборочно скрыть части текста для получения более краткого выхода.

Если указанный голос (или таблицы голосов) не являются неслышимыми (`inaudible`), функция производит поиск речевых кодов для голоса и выстраивает результат в очередь, помещая произносимый текст между `voice-code` и `tts-reset-code` и передавая его речевому серверу.

Использование Aural CSS (ACSS) для стилевого оформления речевого вывода

Сначала я формализовал звуковое форматирование в рамках AsTeR, где правила интерпретации были написаны на специализированном языке, названном языком звукового форматирования — *Audio Formatting Language* (AFL). AFL структурировал параметры, доступные в звуковом пространстве, например высоту используемого голоса, в многомерную область и инкапсулировал состояние движка интерпретатора в виде точки этой многомерной области.

AFL имел блочную структуру, инкапсулирующую текущее состояние интерпретации за счет переменной, имеющей область видимости в пределах лексемы, и предоставлял операторы для перемещения по этой структурированной области. Когда позже эти понятия были спроектированы на декларативный мир HTML и CSS, размерности, составлявшие состояния интерпретации AFL, превратились в параметры Aural CSS, предоставленные в качестве доступных средств в CSS2 (<http://www.w3.org/Press/1998/CSS2-REC>).

Хотя таблицы Aural CSS были задуманы для стилевого оформления деревьев разметки HTML (и вообще XML), они оказались вполне подходящей абстракцией и для построения уровня звукового форматирования `Emacspeak`, наряду с сохранением реализации, независимой от любого заданного TTS-движка.

Определение структуры данных, инкапсулирующей установки ACSS, выглядит следующим образом:

```
(defstruct acss
  family gain left-volume right-volume
  average-pitch pitch-range stress richness punctuations)
```

Emacspeak предоставляет семейство предопределенных *голосовых накладок* (voice overlays) для использования внутри речевых расширений. Голосовые накладки задуманы для *каскадного подключения* в духе Aural CSS. К примеру, установки ACSS, соответствующие voice-monotone, выглядят следующим образом:

```
[c1-struct-acss nil nil nil nil nil 0 0 nil all]
```

Заметьте, что большинство полей в этой структуре acss имеют значение nil, то есть сброшены. Установки создают звуковую накладку, которая:

1. устанавливает pitch в 0 для создания низкого голоса;
2. устанавливает pitch-range в 0 для создания монотонного голоса без интонации.

Эти установки используются как значения свойства personality для звукового форматирования комментариев в режиме обработки текста всех языков программирования. Поскольку его значение является накладкой, оно может эффективно взаимодействовать с другими свойствами звукового отображения. К примеру, если часть комментариев отображается полужирным шрифтом, то эта часть может обладать добавленным значением voice-bolden свойства personality (другой предопределенной накладки); в результате свойство personality приобретет вид таблицы из двух значений: (voice-bolden voice-monotone). В конце концов все это выразится в том, что текст будет произнесен характерным голосом, который выразит оба его аспекта: а именно последовательность слов, которые выделены внутри комментария;

3. устанавливает пунктуацию в all, поэтому при озвучке будут учтены все знаки пунктуации.

Добавление звуковых обозначений

Полноценный визуальный пользовательский интерфейс содержит как текст, так и значки. По аналогии с этим, раз Emacspeak научился выдавать разумную речь, следующий шаг должен был расширить диапазон его звуковой информации и дополнить вывод звуковыми обозначениями.

Звуковые обозначения в Emacspeak представляют собой краткие звуковые фрагменты (длительностью не более двух секунд), которые используются для индикации часто происходящих событий пользовательского интерфейса. К примеру, при каждом сохранении файла пользователем система проигрывает подтверждающий звук. Точно так же открытие или закрытие объекта (любого, от файла до веб-сайта) выдает соответствующее звуковое обозначение. Набор звуковых обозначений постоянно наращивался и охватывал наиболее распространенные события, такие как открытие, закрытие или удаление объектов. В этом разделе рассказывается о том, как эти звуковые обозначения вставлены в выходную систему Emacspeak.

Звуковые обозначения создаются для следующих моментов взаимодействия с пользователем:

- для сигнализации о явных действиях пользователя;
- для добавления дополнительных сигналов к речевому выводу.

Звуковые обозначения, подтверждающие действия пользователя — например что файл был успешно сохранен, — создаются за счет добавления разновидности *after* средства *advice* к различным встроенным компонентам Emacs. Чтобы обеспечить согласующиеся звук и ощущение на всем рабочем столе Emacspeak, такие расширения прикрепляются к коду, который вызывается из многих мест Emacs.

Вот как выглядит пример такого расширения, реализованный за счет фрагмента *advice*:

```
(defadvice save-buffer (after emacspeak pre act)
  "Создание звукового обозначения, если это возможно."
  (when (interactive-p) (emacspeak-auditory-icon 'save-object)
    (or emacspeak-last-message (message "Wrote %s" (buffer-file-name)))))
```

Расширения могут быть реализованы и посредством предоставляемого Emacs перехватчика. Как объяснялось в ранее предоставленном кратком руководстве по *advice*, это средство позволяет расширять или модифицировать существующее программное обеспечение без необходимости модификации основного исходного кода. Emacs сам по себе является расширяемой системой, и у качественно написанного Lisp-кода имеется традиция предоставления соответствующего перехватчика для случаев общего использования. К примеру, Emacspeak прикрепляет звуковую ответную реакцию к исходному механизму подсказок Emacs (к минибуферу Emacs) за счет добавления функции *emacspeak-minibuffer-setup-hook* к имеющейся в Emacs *minibuffer-setup-hook*:

```
(defun emacspeak-minibuffer-setup-hook ()
  "Действие, предпринимаемое при вводе в минибуфер."
  (let ((inhibit-field-text-motion t))
    (when emacspeak-minibuffer-enter-auditory-icon
      (emacspeak-auditory-icon 'open-object))
    (tts-with-punctuations 'all (emacspeak-speak-buffer)))
  (add-hook 'minibuffer-setup-hook 'emacspeak-minibuffer-setup-hook))
```

Это полезный пример использования встроенной расширяемости там, где это доступно. Тем не менее во многих случаях в Emacspeak используется средство *advice*, поскольку в Emacspeak добавление звуковой реакции требуется ко всему, что есть в Emacs, что при первоначальной реализации Emacs не предусматривалось. Поэтому реализация Emacspeak демонстрирует эффективную технологию для обнаружения мест расширения.

Отсутствие в языке программирования средств, подобных *advice*, зачастую усложняет проведение экспериментов, особенно когда дело доходит до обнаружения полезных мест расширения. Именно поэтому разработчики программного обеспечения идут на следующие компромиссные решения:

- делают систему произвольно расширяемой (и произвольно усложненной);
- строят догадки насчет наиболее приемлемых мест расширения и конкретно прописывают их в коде.

Когда места расширения задействованы, эксперименты с новыми местами для расширений требуют переписывания существующего кода, и возникающая инертность зачастую означает, что в течение долгого времени подобные места расширений остаются большей частью не обнаруженными. Имеющееся в Lisp

средство *advice* и его Java-двойник *Aspects* предлагают разработчикам программного обеспечения возможность экспериментировать, не испытывая волнений по поводу неблагоприятного воздействия на существующую основу исходного кода.

Создание звуковых обозначений в процессе произнесения содержимого

В дополнение к использованию звуковых обозначений в качестве реакции на взаимодействие с пользователем Emacspeak использует звуковые обозначения для усиления выразительности произносимого текста. Примеры таких звуковых обозначений включают в себя:

- краткое обозначение начала абзаца;
- звуковое обозначение *mark-object* при пересечении строк исходного кода, на которых установлена контрольная точка.

Звуковые обозначения реализованы за счет прикрепления текстового свойства *emacspeak-auditory-icon* со значением, равным названию звукового обозначения, которое будет запущено на соответствующем тексте.

К примеру, команда установки контрольной точки в пакете отладчика Emacs Grand Unified Debugger (GUD) обработана средством *advice* с целью добавления свойства *emacspeak-auditory-icon* к строке, содержащей контрольную точку. Когда пользователь пересекает такую строку, функция *tts-format-text-andspeak* ставит в очередь звуковое обозначение справа от выходного потока.

Календарь: расширение звукового вывода контекстно-зависимой семантикой

Если подвести промежуточные итоги, то Emacspeak уже способен:

- создавать звуковой вывод из контекста приложения;
- осуществлять звуковое форматирование для расширения диапазона речевой информации;
- дополнять речевой вывод звуковыми обозначениями.

В этом разделе объясняются некоторые конструктивно допустимые расширения.

К созданию Emacspeak как быстрому средству разработки речевого решения для Linux я приступил в октябре 1994 года. Когда в первую неделю ноября я заставил заговорить календарь Emacs, я понял, что фактически создал что-то намного более лучшее, чем все говорящие решения, используемые мною до сих пор.

Календарь – это хороший пример использования определенного типа визуальной компоновки, которая оптимизирована и как наглядное средство, и как передаваемая информация. Рассуждая о датах, мы интуитивно размышляем в понятиях недель и месяцев. Использование табличной раскладки, которая распределяет даты по клеткам, где каждая неделя появляется в собственной

строке, подходит для этого просто идеально. При использовании такой формы размещения человеческий глаз способен быстро перемещаться в календаре по датам, неделям или месяцам, позволяя без труда отвечать на вопросы, вроде: «Какой завтра день недели?» и «Свободна ли у меня третья среда следующего месяца?».

Но следует заметить, что простое озвучивание этого двумерного размещения не переносит эффективность, достигнутую в визуальном контексте в область звуковой интерактивности. Это хороший пример того, как правильная звуковая реакция должна быть генерирована непосредственно из основной сообщаемой информации, а не из ее визуального представления. При создании звукового выхода из визуально отформатированной информации нужно **заново раскрыть основную семантику информации перед тем, как ее озвучивать**.

В отличие от этого, при создании речевого отклика за счет определения *advice*, расширяющего исходное приложение, мы располагаем полным доступом к контексту приложения во время его выполнения. Поэтому вместо того чтобы *строить предположения*, основанные на визуальном размещении, можно просто обучить исходное приложение *озвучивать все правильно!*

Модуль *emacspeak-calendar* превращает календарь Emacs в говорящий за счет определения полезных функций, озвучивающих календарную информацию и обработки с помощью средства *advice* всех команд переходов по календарю для вызова этих функций. Поэтому календарь Emacs ведет себя по-особенному за счет привязки клавиш со стрелками к командам переходов по календарю вместо исходного перемещения курсора, используемого в обычных режимах редактирования. *Emacspeak* приспособливает свое поведение за счет *advice*-обработки команд, присущих календарю для озвучивания соответствующей связанный с календарем информации.

В результате с точки зрения конечного пользователя *все работает как надо*. В обычных режимах редактирования нажатие клавиш-стрелок вверх-вниз приводит к озвучке текущей строки; нажатие этих же клавиш в календаре приводит к переходу на другую неделю и приводит к озвучке текущей даты.

Рассмотрим функцию *emacspeak-calendar-speak-date*, определенную в модуле *emacspeak-calendar*. Заметьте, что она использует все возможности, описанные до сих пор для получения доступа и звукового форматирования соответствующей контекстной информации, полученной из календаря:

```
(defsubst emacspeak-calendar-entry-marked-p( )
  (member 'diary (mapcar #'overlay-face (overlays-at (point)))))

(defun emacspeak-calendar-speak-date( )
  "Озвучивание даты под указателем, когда вызов происходит в режиме календаря."
  (let ((date (calendar-date-string (calendar-cursor-to-date t))))
    (cond
      ((emacspeak-calendar-entry-marked-p) (tts-speak-using-voice mark-personality
date))
      (t (tts-speak date)))))
```

Emacs помечает даты, имеющие дневниковые записи, специальной накладкой. В предыдущем определении вспомогательная функция *emacspeak-calendar-entry-marked-p* проверяет наличие этой накладки для создания предиката, который может быть использован для тестирования наличия даты у дневниковой

записи. Функция `emacspeak-calendar-speak-date` использует этот предикат для решения, нужно ли выводить эту дату другим голосом; даты, имеющие календарные записи, озвучиваются с использованием голоса `markpersonality`. Заметьте, что функция `emacspeak-calendar-speak-date` получает доступ к содержимому календаря в процессе работы программы при следующем вызове:

```
(calendar-date-string (calendar-cursor-to-date t))
```

Функция `emacspeak-calendar-speak-date` вызывается из *advice*-определений, прикрепленных ко всем функциям переходов по календарю. Вот как выглядит *advice*-определение для функции `calendar-forward-week`:

```
(defadvice calendar-forward-week (after emacspeak pre act)
  "Озвучивание даты."
  (when (interactive-p) (emacspeak-speak-calendar-date)
    (emacspeak-auditory-icon 'large-movement)))
```

Это *after*-разновидность *advice*, поскольку нам нужно, чтобы звуковая реакция получалась *после* того, как обычная команда перехода завершит свою работу.

Тело *advice*-определения сначала вызывает функцию `emacspeak-calendar-speak-date` для озвучивания даты под курсором; затем оно вызывает `emacspeak-auditory-icon` для выдачи короткого звука, оповещающего, что переход был успешно осуществлен.

Беспроblemный доступ к интерактивной информации

Теперь, имея в своем распоряжении все необходимые средства для генерации полноценного звукового выхода, говорящие приложения Emacs, используя средство *advice*, принадлежащее Emacs Lisp, требуют на удивление малого объема специализированного кода. С помощью уровня TTS и ядра Emacspeak, ведущих обработку сложных компонентов при создании высококачественного выхода, говорящие расширения концентрируются исключительно на специализированной семантике конкретных приложений; это приводит к простому и несомненно *красивому* коду. В этом разделе данная концепция иллюстрируется с помощью нескольких отобранных примеров, взятых из богатого набора Emacspeak-инструментария, предназначенного для доступа к информации.

В то же самое время, когда я приступил к созданию Emacspeak, в компьютерном мире произошла куда более существенная революция: всемирная паутина – World Wide Web превратилась из инструмента академических исследований в господствующий форум для решения повседневных задач. Это было в 1994 году, когда написание браузера было сравнительно легкой задачей. Усложнения, постоянно добавляемые в Интернет последующие 12 лет, зачастую имели тенденцию затенять тот факт, что Интернет в своей основе по-прежнему является довольно простой конструкцией, где:

- создатели содержимого публикуют веб-ресурсы по адресам, доступным через URI;

- URI-адресуемое содержимое может извлекаться по открытым протоколам;
- извлекаемое содержимое представляет собой HTML, вполне понятный язык разметки.

Заметьте, что наметки основной архитектуры практически ничего не говорят, каким образом содержимое становится доступным конечному пользователю. В середине 1990-х годов Интернет развивался в направлении бурного роста визуальной интерактивности. Коммерческий Интернет, склонный к яркой визуальной интерактивности, все больше и больше отходил от простой интерактивности, основанной на данных, которой отличались ранние веб-сайты. К 1998 году я понял, что в Интернете имеется множество полезных интерактивных веб-сайтов; к своему сожалению, я также понял, что пользуюсь все меньшим количеством этих веб-сайтов, экономя время на завершение задач, в которых используется речевой вывод.

Это привело меня к созданию набора веб-ориентированного инструментария в рамках Emacspeak, обращенного к основам веб-взаимодействия. Emacs уже имел возможность отображения простого кода HTML в виде интерактивных гипертекстовых документов. По мере усложнения Интернета, Emacspeak пополнился коллекцией интерактивных мастеров, надстроенных над возможностью Emacs по отображению HTML, которая постепенно разбиралась со сложностями веб-взаимодействия для создания озвученного интерфейса, позволяющего пользователю быстро и беспроблемно прослушивать желаемую информацию.

Элементарный HTML с Emacs W3 и Aural CSS

Emacs W3 – это простейший веб-браузер, впервые выпущенный в середине 1990-х годов. Вскоре в Emacs W3 появилась реализация каскадных таблиц стилей – CSS (Cascading Style Sheets), что послужило основой для первой реализации звуковых таблиц стилей – Aural CSS, которые были выпущены после того, как я в феврале 1996 года написал их проект. Emacspeak озвучил Emacs W3 посредством модуля `emacspeak-w3`, который обеспечивал следующие расширения.

- Раздел `aural media` в исходных таблицах стиля для Aural CSS.
- Средство `advice`, добавленное ко всем интерактивным командам для выдачи звуковой реакции.
- Специальные шаблоны для распознавания и игнорирования молчанием декоративных изображений на веб-страницах.
- Звуковое отображение полей HTML-форм вместе с их надписями, которые были заложены в основе конструкции элемента `label` в HTML 4.
- Контекстно-зависимые правила отображения для элементов управления HTML-формы. К примеру, заданной группы переключателей для ответа на вопрос:

Вы согласны?

Emacspeak расширяет Emacs W3 для выдачи речевого сообщения в форме:
Переключатель Вы согласны? находится в состоянии Да.

и:

Щелкните здесь, чтобы изменить состояние переключателя Вы согласны?
с Да на Нет.

- Разновидность before средства *advice* определена для Emacs W3 функции `w3-parse-buffer`, которая занимается запрошенным пользователем преобразованием XSLT в HTML-страницы.

Модуль **emacspeak-websearch** для целенаправленного поиска

К 1997 году интерактивные веб-сайты Интернета варьировали от поисковой системы Altavista до Yahoo! Maps для оперативного определения направлений движения, требующих от пользователя прохождения вполне наглядного процесса, который включал:

1. заполнение набора полей формы;
2. отправку получившейся формы;
3. разыскивание результатов в полученной составной HTML-странице.

Первый и третий шаги при использовании речевого вывода требовали определенного времени. Сначала мне нужно было определить местоположение различных полей формы на визуально заполненной странице и пробраться через завалы сложного статейного материала, прежде чем я нашел ответ.

Заметьте, что с точки зрения разработки программного обеспечения эти шаги четко проецировались на перехваты до действия — *pre-action* и после действия — *post-action*. Поскольку веб-взаимодействие придерживается очень простой архитектуры, основанной на URI, шаг до действия (*pre-action*), приглашающий пользователя в нужные места ввода информации, должен быть вынесен за пределы веб-сайта и помещен в небольшой фрагмент кода, запускаемый локально; это избавит пользователя от необходимости открывать первоначально запущенную страницу и искать различные поля ввода.

Таким же образом, шаг после действия (*post-action*) по розыску фактических результатов во всей остальной бесполезной информации на полученной странице также может быть передан программному обеспечению.

И наконец, заметьте, что даже при том что все эти шаги до и после действия специфичны для конкретных веб-сайтов, общая конструктивная схема может быть универсальной. Осознание этого обстоятельства привело к созданию модуля *emacspeak-websearch*, являющегося коллекцией проблемно-ориентированного веб-инструментария, который:

1. выдает пользователю приглашение;
2. создает соответствующий URI и наполняет этот URI содержимым;
3. фильтрует результат перед отображением соответствующего содержимого в Emacs W3.

Вот как выглядит инструмент *emacspeak-websearch* для вызова направлений из Yahoo! Maps:

```
(defsubst emacspeak-websearch-yahoo-map-directions-get-locations ( )
  "Удобная функция для приглашения к составлению и для составления части
  маршрута."
  (concat
    (format "&newaddr=%s"
            (emacspeak-url-encode (read-from-minibuffer "Start Address: ")))
    (format "&newcsz=%s"
            (emacspeak-url-encode (read-from-minibuffer "City/State or Zip:")))
    (format "&newtaddr=%s"
            (emacspeak-url-encode (read-from-minibuffer "Destination Address: ")))
    (format "&newtcsz=%s"
            (emacspeak-url-encode (read-from-minibuffer "City/State or Zip:")))))
(defun emacspeak-websearch-yahoo-map-directions-search (query )
  "получение направления движения от Yahoo."
  (interactive
    (list (emacspeak-websearch-yahoo-map-directions-get-locations)
          (emacspeak-w3-extract-table-by-match
            "Start"
            (concat emacspeak-websearch-yahoo-maps-uri query))))
```

Рассмотрим краткое пояснение предыдущего кода.

Pre-action

Функция `emacspeak-websearch-yahoo-map-directions-get-locations` выдает пользователю приглашение обозначить начальный и конечный пункты маршрута. Заметьте, что функция жестко фиксирует имена параметров запроса, используемых Yahoo! Maps. Казалось бы, это похоже на упущение, которое приведет к возникновению ошибки. Но на самом деле это «упущение» не вызывает негативных последствий со временем своего первого определения в 1997 году. Причина очевидна: поскольку веб-приложение опубликовало набор запрашиваемых параметров, эти параметры стали жестко зафиксированными в целом ряде мест, включая HTML-страницы на исходном веб-сайте. Зависимость от названия параметра может представляться разработчику архитектуры программного обеспечения, привыкшего к структурированному, исходящему API, чем-то непрочным, но использование подобных параметров URL для определения восходящих веб-служб приводит к идеи таких веб-API, как RESTful.

Искомое содержимое

URL для искомых направлений создан путем объединения пользовательского ввода с базовым *URI* для Yahoo! Maps.

Post-action

Полученный таким образом URI передается функции `emacspeak-w3-extract-table-by-match` вместе с поисковым шаблоном `Start`, чтобы:

- найти информационное наполнение, используя Emacs W3;
- применить XSLT-преобразование для извлечения таблицы, содержащей `Start`;
- отобразить эту таблицу, используя средства HTML-форматирования, имеющиеся в Emacs W3.

В отличие от параметров запроса, формат страницы результатов *изменяется*, в среднем, раз в году. Но поддержание этого инструмента в соответствии

с текущими установками Yahoo! Maps сводится к обслуживанию той части утилиты, которая относится к post-action. За более чем восемь лет использования мне приходилось вносить изменения около шести раз, но учитывая, что основная платформа обеспечивает многие инструменты фильтрации страницы, выдаваемой в качестве результата, реальное количество строк кода для каждого изменения формата совсем невелико.

Функция `emacspeak-w3-extract-table-by-match` использует XSLT-преобразование, которое фильтрует документ, чтобы вернуть таблицы, содержащие определенную схему поиска. Для данного примера функция составляет следующее XPath-выражение:

```
(/descendant::table[contains(.. Start)])[last( )]
```

Она фактически извлекает список таблиц, в которых содержится строка Start, и возвращает последний элемент этого списка.

Через семь лет после того, как была написана эта утилита, компания *Google* в феврале 2005 года запустила Google Maps, вызвав большой ажиотаж. Многие блоги Интернета поместили Google Maps под микроскоп и быстро обнаружили параметры запроса, используемые этим приложением. Я воспользовался этим для создания соответствующего инструмента Google Maps в Emacspeak, который обеспечивал аналогичную функциональность. Инструмент, предназначенный для Google Maps, оставляет у пользователя более приятные впечатления, поскольку начальная и конечная точки маршрута могут быть указаны в одном и том же параметре. Вот как выглядит код мастера для Google Maps:

```
(defun emacspeak-websearch-emaps-search (query &optional use-near)
  "Осуществление EmapSpeak-поиска. Запрос - на простом английском языке."
  (interactive
   (list
    (emacspeak-websearch-read-query
     (if current-prefix-arg
         (format "Find what near %s: "
                 emacspeak-websearch-emapspeak-my-location)
         "EMap Query: "))
     current-prefix-arg))
  (let ((near-p :: определение типа запроса
        (unless use-near
          (save-match-data (and (string-match "near" query) (match-end 0))))))
        (near nil)
        (uri nil))
    (when near-p :: определение места из запроса
      (setq near (substring query near-p))
      (setq emacspeak-websearch-emapspeak-my-location near))
    (setq uri
          (cond
            (use-near
             (format emacspeak-websearch-google-maps-uri
                     (emacspeak-url-encode
                      (format "%s near %s" query near))))
            (t (format emacspeak-websearch-google-maps-uri
                      (emacspeak-url-encode query))))))
  (add-hook 'emacspeak-w3-post-process-hook 'emacspeak-speak-buffer)
  (add-hook 'emacspeak-w3-post-process-hook
```

```
#'(lambda nil
  (emacspeak-pronounce-add-buffer-local-dictionary-entry
   "#240:mi" " miles")))
(browse-url-of-buffer
 (emacspeak-xslt-xml-uri
  (expand-file-name "km12html.xsl" emacspeak-xslt-directory)
  uri)))
```

Краткое пояснение.

1. Разбор введенной информации, чтобы решить, что это, направление или запрос на поиск.
2. Если это запрос на поиск, кэшировать местонахождение пользователя для дальнейшего использования.
3. Построение URI для поиска результатов.
4. Просмотр результатов отфильтрованного содержимого URI через XSLT-фильтр km12html, который преобразует найденный контекст в простой гипертекстовый документ.
5. Установка специального озвучивания, чтобы сокращение `mi` произносилось как «miles» (мили).

Заметьте, что как и раньше, большая часть кода сконцентрирована на задачах, характерных для приложения. Полноценный речевой вывод вырабатывается за счет создания результатов в виде HTML-документа, имеющего правильную структуру, с соответствующими правилами Aural CSS, производящими представление с отформатированным звучанием.

Командная строка Интернета и шаблоны URL

С ростом в Интернете количества доступных служб в начале 2000-х годов появился другой полезный шаблон: веб-сайты стали создавать интеллектуальное взаимодействие на стороне клиента с использованием JavaScript. Одним из типичных применений подобных сценариев стало построение URL на стороне клиента для доступа к определенным частям содержимого на основе пользовательского ввода. К примеру, высшая бейсбольная лига составляет URL для извлечения результатов заданной игры, объединяя вместе даты, названия принимающей и гостящей команды, а национальное радио NPR создает URL, объединяя вместе дату с кодом программы заданной радиотрансляции.

Чтобы получить быстрый доступ к таким службам, я в конце 2000 года добавил модуль `emacspeak-url-template`. Этот модуль стал мощным компаньоном модулю `emacspeak-websearch`, который был рассмотрен в предыдущем разделе. Работая вместе, эти модули превращают минибуфер Emacs в мощную сетевую командную линию, обеспечивающую быстрый доступ к сетевому содержимому.

Многие веб-службы требуют от пользователя указания даты. Для получения содержимого каждый может воспользоваться датой по умолчанию, используя пользовательский календарь. Поэтому инструментарий Emacspeak для прослушивания NPR-программ или получения результатов матчей высшей лиги настроен по умолчанию на использование даты, на которую указывает курсор, когда она вызывается из буфера календаря Emacs.

Шаблоны URL в Emacspeak реализованы с использованием следующей структуры данных:

```
(defstruct (emacspeak-url-template (:constructor emacspeak-ut-constructor))
  name      :: Название в человеческом восприятии
  template   :: строка шаблона URL
  generators :: список генератора параметров
  post-action :: действие, выполняемое после открытия
  documentation :: документация ресурса
  fetcher)
```

Пользователи вызывают шаблоны URL посредством команды Emacspeak emacspeak-url-template-fetch, которая запрашивает шаблон URL и:

1. ведет поиск названного шаблона;
2. запрашивает пользователя, вызывая определенный generator;
3. применяет функцию Lisp format к строке шаблона и к собранным аргументам для создания окончательного варианта URI;
4. вызывает любые *последействия* (post actions), осуществляемые после передачи содержимого;
5. применяет указанный сборщик (fetcher) для отображения содержимого.

Применение этой структуры лучше всего объясняется на примере. Рассмотрим шаблон URL для прослушивания радиопрограмм NPR:

```
(emacspeak-url-template-define
  "NPR по запросу"
  "http://www.npr.org/dmg/dmg.php?prgCode=%s&showDate=%s&segNum=%s&mediaPref=RM"
  (list
    #'(lambda () (upcase (read-from-minibuffer "Program code:")))
    #'(lambda ()
        (emacspeak-url-template-collect-date "Date:" "%d-%b-%Y"))
    "Segment:")
  nil: no post actions
```

"Трансляция NPR-программ по запросу.

Программа указывается в виде принадлежащего ей кода:

ME	Morning Edition - утренний выпуск
ATC	All Things Considered - все сообщения
day	Day To Day - день за днем
newsnotes	News And Notes - новости и комментарии
totn	Talk Of The Nation - национальные новости
fa	Fresh Air - свободный эфир
wesat	Weekend Edition Saturday - еженедельный субботний выпуск
wesun	Weekend Edition Sunday - еженедельный воскресный выпуск
fool	The Motley Fool - всякие глупости

Каждая передача указывается двумя цифрами - указание пустого значения приводит к воспроизведению всей программы."

```
#'(lambda (url)
  (funcall emacspeak-media-player url 'play-list)
  (emacspeak-w3-browse-xml-url-with-style
    (expand-file-name "smil-anchors.xsl" emacspeak-xslt-directory)
    url)))
```

В данном примере заказной сборщик — fetcher выполняет два действия:

1. запускает медиапроигрыватель для начала прослушивания аудиопотока;
2. фильтрует соответствующий SMIL-документ с помощью XSLT-файла smil-anchors.xsl.

Появление программ считывания веб-каналов

Когда я создал модули `emacspeak-websearch` и `emacspeak-url-template`, программе Emacspeak требовалось буквально «добывать» с экрана HTML-страницы значимую информацию для ее дальнейшей озвучки. Но по мере усложнения Интернета необходимость в легком способе пробраться сквозь внешнее представление страниц к их реальному информационному наполнению приобрела более широкое значение, чем невизуальный доступ. Даже пользователи, имеющие возможность работать со сложными визуальными интерфейсами, испытывали серьезную информационную перегрузку. Это привело к появлению веб-каналов RSS и Atom и созданию соответствующего программного обеспечения для чтения этих веб-каналов.

Эти события положительным образом повлияли на программную основу Emacspeak. В течение последних нескольких лет код стал *более красивым*, поскольку я постепенно удалил логику добычи информации с экрана и заменил ее на непосредственный доступ к информационному наполнению. В качестве примера можно привести Emacspeak URL-шаблон для извлечения прогноза погоды для заданного города и штата:

```
(emacspeak-url-template-define
  "Погодный rss-канал с веб-сайта wunderground"
  "http://www.wunderground.com/auto/rss_full%{s}.xml?units=both"
  (list "Штат/Город, например: MA/Boston") nil
  "Извлечение погодного RSS-канала для заданного города и штата."
  'emacspeak-rss-display)
```

А вот как выглядит URL-шаблон для извлечения новостей Google News, получаемых через веб-канал Atom:

```
(emacspeak-url-template-define
  "Поиск Google News"
  "http://news.google.com/news?hl=en&ned=tus&q=%{s}&btnG=Google+Search&output=atom"
  (list "Поиск новостей для: ") nil "Поиск новостей Google."
  'emacspeak-atom-display )
```

Оба этих инструмента используют все средства, предоставляемые модулем `emacspeak-url-template`, поэтому дополнительной затраты существенных усилий не потребовалось. И наконец, обратите внимание, что в расчете на стандартизацию форматов таких веб-потоков, как RSS и Atom, в этих шаблонах теперь мало что осталось от вставок, специфических для веб-сайтов, по сравнению с более старыми инструментальными средствами вроде мастера Yahoo! Maps, в котором жестко задан шаблон получаемой страницы.

Краткий отчет

Emacspeak задумывался как законченный, невизуальный пользовательский интерфейс для решения повседневных компьютерных задач. Чтобы быть *законченной*, система нуждается в предоставлении прямого доступа ко всем аспектам вычислений на настольных рабочих станциях. Чтобы сделать возможным быстрое *невизуальное взаимодействие*, система должна была рассматривать речевой

вывод и звуковые средства в качестве компонентов первого класса — то есть простого озвучивания отображаемой на экране информации было недостаточно.

Чтобы предоставить пользователю *полноценный звуковой настольный компьютер*, заданная среда должна быть широко используемой и полностью расширяемой структурой взаимодействия. Чтобы быть в состоянии не только озвучивать содержимое экрана, система должна была встроить интерактивные речевые возможности в различные приложения.

И наконец, это должно быть сделано без модификации исходного кода любого из основных приложений; проект не мог себе позволить разветвления приложений во имя добавления невизуального взаимодействия, поскольку я хотел ограничиться задачей поддержки речевых расширений.

Чтобы выполнить все эти конструктивные требования, я выбрал в качестве среды пользовательского взаимодействия Emacs. В качестве структуры взаимодействия у Emacs были преимущества поддержки со стороны очень широкого сообщества разработчиков. В отличие от других популярных структур взаимодействия, доступных в 1994 году, когда я начинал работу над проектом, существенное преимущество достигалось за счет открытости программной среды. (Теперь, 12 лет спустя, такие же возможности предоставляет Firefox.)

Широчайшая гибкость, предоставляемая Emacs Lisp как языком расширения, была важнейшей предпосылкой в озвучивании различных приложений. Открытость программного кода платформы также была очень важным обстоятельством; даже при твердом решении не модифицировать существующий код, возможность изучать реализацию различных приложений превращало их озвучивание во вполне посильную задачу. И наконец, доступность высококачественной реализации средства *advice* для Emacs Lisp (учтите, что имеющееся в Lisp средство *advice* было главным мотивационным фактором для аспектно-ориентированного программирования) дала возможность озвучить приложения, созданные в Emacs Lisp без внесения изменений в их оригинальный исходный код.

Emacspeak — это прямое следствие совпадения ранее обозначенных потребностей и возможностей, предоставленных Emacs в качестве структуры взаимодействия с пользователем.

Укрощение сложности кода в течение длительного периода времени

За период более 12 лет код Emacspeak получил свое развитие. Исключение составили первые шесть недель разработки, когда основа кода была разработана и поддержана с использованием непосредственно Emacspeak. В этом разделе подводятся итоги, извлеченные из некоторых уроков по укрощению сложности кода в течение длительного периода времени.

За все время своего существования Emacspeak всегда оставался проектом свободного времени. Просматривая основу кода сквозь призму времени, я полагаю, что он оказал существенное влияние на развитие системы. Работая над крупными и сложными программными системами в полную силу, разработчик может себе позволить полностью сосредоточиться на кодовой основе в течение

вполне разумного периода времени – например, от 6 до 12 недель. В результате получается добротно разработанный код, закладывающий *крепкую* основу всей программы.

Несмотря на самые благие намерения, все это может привести к разработке кода, в котором со временем становится очень трудно разобраться. Крупные программные системы, создаваемые единственным разработчиком, сосредоточенным только на этом проекте в течение нескольких лет, – большая редкость; такая форма персональной целеустремленности приводит обычно к быстрому угасанию проекта!

В отличие от этого, Emacspeak является крупной программной системой, над которой один разработчик трудился свыше 12 лет, но только в свое свободное время. Вследствие того что система разрабатывалась единолично в течение нескольких лет, программная основа приобрела вполне естественную «разветвленность». Обратите внимание на итоговое распределение файлов и строк кода, показанное в табл. 31.1.

Таблица 31.1. Итоговые характеристики программной основы Emacspeak

Уровень	Файлы	Строки	Процентное отношение
Ядро TTS	6	3866	6,0
Ядро Emacspeak	16	12174	18,9
Расширения Emacspeak	160	48339	75,0
Итого	182	64379	99,9

Таблица 31.1 высвечивает следующие моменты:

- Ядро TTS, отвечающее за высококачественный речевой вывод, занимает 6 из 182 файлов и составляет 6 % программной основы.
- Ядро Emacspeak, предоставляющее высокоуровневые службы для расширений Emacspeak вдобавок к озвучанию всех основных функциональных возможностей Emacs, занимает 16 файлов и составляет около 19 % программной основы.
- Вся остальная система распределена по 160 файлам, которые могут быть независимо друг от друга улучшены (или испорчены), не оказывая влияния на всю остальную систему. Множество модулей наподобие `emacspeak-urltemplate` являются разветвленными сами по себе – то есть отдельный URL-шаблон может быть модифицирован, не оказывая влияния на любые другие URL-шаблоны.
- Применение средства *advice* сокращает размер кода. Программная основа Emacspeak, в которой содержится примерно 60 000 строк кода на языке Lisp, является лишь частью от размера основной озвученной системы. Приблизительный подсчет, сделанный в декабре 2006 года, показал, что Emacs 22 содержит более миллиона строк Lisp-кода; кроме того, речевые возможности, предоставляемые Emacspeak, распространяются на большое количество приложений, изначально не связанных с Emacs.

Вывод

На основе реализации и использования Emacspeak можно сделать следующие краткие выводы.

- Имеющееся в Lisp средство *advice* и его объектно-ориентированный эквивалент – аспектно-ориентированное программирование – является весьма эффективным средством решения перекрестных проблем – например озвучки визуального интерфейса.
- *Advice* является мощным средством для обнаружения потенциальных мест расширений сложных программных систем.
- Сосредоточенность на базовой веб-архитектуре и надежда на информационно-ориентированную сеть, поддерживаемую стандартизацией протоколов и форматов, приводит к созданию мощного озвученного доступа к веб-ресурсам.
- Концентрация на впечатлениях конечного пользователя в противовес отдельным графическим элементам взаимодействия, вроде ползунков прокрутки и элементов управления деревом, приводит к созданию высокоэффективной невизуальной рабочей среды.
- Визуальное взаимодействие во многом полагается на способность человеческого глаза быстро просматривать графическое отображение информации. Эффективное невизуальное взаимодействие требует передачи части этих функций компьютеру, поскольку прослушивание большого объема информации занимает много времени. Поэтому различные формы поиска играют существенную роль для успешного осуществления невизуального взаимодействия в масштабах от самого незначительного (такого как производимый в Emacs поэтапный поиск нужного элемента в локальном документе) до самого большого (такого как организованный в Google быстрый поиск нужного документа в глобальной сети).
- Визуальная сложность, раздражающая пользователей, имеющих возможность использования замысловатых визуальных интерфейсов, становится непреодолимым препятствием для невизуального взаимодействия. И наоборот, инструментальные средства, первоначально появляющиеся в невизуальной среде, в конечном итоге становятся доминирующими, когда неудобства сложных визуальных интерфейсов превышают определенные пределы. Исходя из опыта применения Emacspeak, можно привести в пример два подобных случая:
 - веб-каналы RSS и Atom подменили необходимость добычи информации с экрана простым извлечением значимой информации, в частности статейных заголовков;
 - использование в 2000 году в Emacspeak технологии XSLT для фильтрации информационного наполнения можно сравнить с внедрением в 2005 году расширения Greasemonkey для использования на веб-страницах клиентского JavaScript, выполняющего отдельные задачи.

Благодарности

Emacspeak не мог появиться без Emacs и всегда энергичного сообщества разработчиков Emacs, которое дало возможность делать внутри Emacs все, что угодно. Реализация Emacspeak не состоялась бы без великолепной реализации средства *advice* для Emacs Lisp, выполненной Хансом Чалупски (Hans Chalupsky).

Проект *libxslt*, являющийся частью проекта GNOME, помог вдохнуть новую жизнь в разработанный Уильямом Перри (William Perri) браузер Emacs W3; этот браузер был одним из самых первых движков отображения HTML, но код не подвергался обновлению свыше восьми лет. То, что программная основа W3 до сих пор используется и наращивается, является ярким доказательством той гибкости и мощности, которая предоставляется языком программирования Lisp.

32 Код в развитии

*Лаура Уингерд (Laura Wingerd)
и Кристофер Сейвальд (Christopher Seiwald)*

Главное в том, что каждый удачный образец программного обеспечения является долгожителем и работает на поколения программистов и проектировщиков...

Бьери Струструп (Bjarne Stroustrup)

На ранней стадии замысла этой книги Грэг Уилсон (Greg Wilson) спрашивал участников ее создания, станет ли Идеальный код подходящим для нее названием. Он писал всем нам: «Главным предметом рассмотрения является проектирование программного обеспечения и архитектура, а не программный код как таковой».

Но эта глава *именно* о программном коде. Она посвящена не тому, что он делает, и не его красоте, а тому, как он *выглядит*: а именно, как определенные, очевидные для людей характерные особенности частей программы позволяют осуществлять последовательное сотрудничество. Эта глава о красоте «кода в развитии».

То, что вам предстоит прочесть, в значительной степени позаимствовано из статьи Кристофера Сейвальда (Christopher Seiwald) «The Seven Pillars of Pretty Code»¹ («Семь основных принципов красивого кода»). Вот как выглядят эти принципы в кратком изложении.

- Применять «книгообразный» вид.
- Использовать однообразный вид для одинакового кода.
- Избегать отступов.
- Высвобождать блоки кода.
- Комментировать блоки кода.
- Избавиться от всего лишнего.
- Учитывать сочетаемость с существующим стилем.

¹ Статья доступна на веб-сайте Perforce: <http://www.perforce.com/perforce/papers/prettycode.html>.

Все это похоже на простое соглашение по программированию, но значение этих принципов куда серьезнее простого соглашения. Это объективное олицетворение практики программирования, выработанное с учетом дальнейшего развития продукта.

В этой главе мы посмотрим, как «Семь принципов» послужили основанием для той части программного кода, которая была составляющей коммерческой программной системы на протяжении более 10 лет. Речь пойдет о DiffMerge, компоненте Perforce, системы управления конфигурацией программного обеспечения. Работа DiffMerge состоит в создании классического трехстороннего объединения, в сравнении двух версий текстового файла («сторона 1» и «сторона 2») с эталонной версией («основания»). В выходной информации чередуются строки входных файлов с пометками о конфликтующих строках. Если вы пользовались Perforce, то видели DiffMerge в работе, при использовании команды `r4 resolve`, и в графическом инструментарии Perforce, предназначенному для объединения.

Первоначальная версия DiffMerge была написана в 1996 году. Несмотря на простоту своего предназначения, функция трехстороннего объединения текста оказалась очень запутанной. Это плавильный тигель для особых случаев, возникающих из-за особенностей пользовательских интерфейсов, кодировок символов, языков программирования и самих программистов («Здесь нет конфликта». «Нет, есть». «Нет, его здесь нет!»).

С годами DiffMerge стал в Perforce Software базой для существенных усовершенствований. Поэтому нам недостаточно, что DiffMerge — это простой и правильный образец кода. Он должен быть тем самым образцом, который «хорошо проигрывает» тот инструментарий, который мы используем для программирования, отладки и управления изменениями. И он должен стать тем образцом кода, в котором предвидятся изменения.

Путь от первой реализации DiffMerge к его сегодняшней форме был, мягко говоря, не гладким. Видимо, не стало совпадением, что чем больше мы отступали от семи основополагающих принципов, тем более каменистым становился этот путь. Позже в этой главе мы покажем некоторые колдобыны (и одну крупную аварию), которые встречались на пути десятилетнего странствия DiffMerge.

Но все хорошо, что хорошо кончается. Сегодня DiffMerge, переизданный на веб-сайте <http://www.perforce.com/beautifulcode>, — стабильный компонент, легко восприимчивый к изменениям. Он служит примером того, как программирование с прицелом на будущие изменения может выдавать красивый образец кода в развитии.

Применение «книгообразного» вида

«Семь основных принципов красивого кода» описывают те нормативы, которые мы используем в Perforce Software. Семь принципов¹ — не единственный

¹ В нашем офисе мы не называем их «Семью принципами». Фактически мы не воспринимаем их чем-то обособленным от наших специфических языковых или компонентных нормативов программирования. Но если убрать все остальное, то останутся только они.

норматив, которого мы придерживаемся, к тому же они подходят не ко всем нашим проектам разработки программного обеспечения. Мы применяем их к таким компонентам, как DiffMerge, где один и тот же код, скорее всего, будет применяться в нескольких параллельно поддерживаемых версиях и модифицироваться множеством программистов. «Семь принципов» в большинстве случаев позволяют сделать код более понятным для программистов, которым приходится его читать.

Возьмем, к примеру, один из этих принципов — применение «книгообразного» вида. Текст в книгах и журналах набирается по колонкам, ширина которых обычно меньше ширины самой страницы. Зачем это делается? Да затем, что узкая колонка сокращает охватываемое взглядом пространство при движении глаз вперед-назад — чтение становится проще, когда глаза меньше работают. Чтение также упрощается, когда то, что мы уже читаем, и то, что собираемся прочесть, находится в поле нашего зрения. Исследования показали, что наши глаза меняют фокусировку от слова к слову, а мозг может воспринять ключевую информацию из окружения, имеющего неясные очертания. Чем больше наш мозг может собрать «опережающих признаков» из очертаний зрительной периферии, тем точнее он может направить наш взгляд для максимального понимания текста.

Исследования также показали, что от длины строки зависит разница между скоростью чтения и скоростью понимания прочитанного. Более длинные строки могут быть прочитаны быстрее, а более короткие проще понять.

Разбитый на части текст также проще воспринимается, чем длинная колонка. Поэтому колонки в книгах и журналах поделены на абзацы. Абзацы, строфы, списки, врезки и сноски — это своеобразные «маркеры общения» с текстом, которые «говорят» нашему мозгу: «Ну что, вы уже вникли в суть всего до сих пор изложенного? Отлично, давайте продолжим».

Разумеется, программный код, строго говоря, не является простым текстом, но для облегчения чтения к нему применяются те же принципы. Книгообразный код, то есть код, оформленный наподобие книжных колонок и разбиений, легче понять.

Книгообразность — это более широкое понятие, чем простое соблюдение правила коротких строк. Это различие между этим кодом:

```
if( bf->end == bf->Lines() && lf1->end == lf1->Lines( ) &&
   lf2->end == lf2->Lines( ) ) return( DD_EOF );
```

и вот этим:

```
if( bf->end == bf->Lines( ) &&
   lf1->end == lf1->Lines( ) &&
   lf2->end == lf2->Lines( ) )
      return( DD_EOF );
```

Второй фрагмент кода взят из DiffMerge. Когда мы его читаем, наш мозг воспринимает рассматриваемый логический контекст, а глазам не приходится для этого слишком далеко переводить взгляд из стороны в сторону. (Также важен и факт существования определенного визуального образа, созданного за счет

разрывов строки, совсем скоро мы доберемся и до этого.) Благодаря книгообразному формату второй фрагмент кода воспринимается намного легче первого.

Однаковое должно выглядеть однообразно

Фрагмент DiffMerge в предыдущем разделе иллюстрирует также другой принцип понятной записи: код, имеющий похожее содержание, должен выглядеть однообразно. Мы это поймем, просмотрев код DiffMerge. Например:

```
while( d.diffs == DD_CONF && ( bf->end != bf->Lines( ) ||  
    lf1->end != lf1->Lines( ) ||  
    lf2->end != lf2->Lines( ) ) )
```

Этот пример демонстрирует, как разрывы строки могут создавать визуальный образ, облегчающий нашему мозгу распознавание логической структуры. Мы с первого взгляда можем сказать, что три из четырех тестов в этом операторе while по своей сути одинаковы.

А вот еще один пример, когда одинаковое выглядит однообразно. В нем иллюстрируется прием написания кода, позволяющий нашему мозгу успешно провести операцию под названием «одна из частей не похожа на другие»:

```
case MS_BASE: /* dumping the original */  
  
if( selbits = selbitTab[ DL_BASE ][ diffDiff ] )  
{  
    readFile = bf;  
    readFile->SeekLine( bf->start );  
    state = MS_LEG1;  
    break;  
}  
  
case MS_LEG1: /* dumping leg1 */  
  
if( selbits = selbitTab[ DL_LEG1 ][ diffDiff ] )  
{  
    readFile = lf1;  
    readFile->SeekLine( lf1->start );  
    state = MS_LEG2;  
    break;  
}  
  
case MS_LEG2: /* dumping leg2 */  
  
if( selbits = selbitTab[ DL_LEG2 ][ diffDiff ] )  
{  
    readFile = lf2;  
    readFile->SeekLine( lf2->start );  
}  
  
state = MS_DIFFDIFF;  
break;
```

Даже если вы не знаете, в чем смысл этого кода, совершенно нетрудно, к примеру, понять, что во всех трех случаях устанавливаются значения для `readfile` и `state`, но только в третьем случае значение `state` устанавливается за рамками условий. Написавший этот код программист уделил внимание тому, чтобы одинаковое выглядело однообразно; те, кто будет читать этот код позже, смогут сразу понять, где здесь важная логическая особенность.

Опасность отступов

Нас всех научили, что нужно использовать отступы, чтобы показать глубину вложений логических блоков кода. Чем глубже вложение, тем сильнее смещается вправо на странице вложенный код. Подобное форматирование кода — хорошая идея, но не потому, что она облегчает чтение кода.

Во всяком случае код, размещенный с глубокими отступами, читается труднее. Важная логика прижата вправо, низведена окружающими ее наслоениями кода `if-then-else` едва ли не до роли второстепенной сноски, тогда как незначительные проверки, применяющиеся во внешних блоках, кажутся необоснованно важными. Итак, несмотря на то что отступы весьма полезны для отображения начала и конца блока, они не облегчают нам понимание кода.

Разумеется, что не отступы создают главную опасность: она заключается во вложенности. Вложенный код заставляет человека напрягаться, нанося ущерб простоте и ясности понимания. Вряд ли Эдвард Тафт (Edward Tufte) хотел сделать комплимент, когда писал следующее: «Иногда иерархия маркеров [в PowerPoint] настолько сложна и имеет столь глубокие вложения, что напоминает компьютерный код». В книге «Code Complete» (см. раздел «Дополнительная информация» в конце главы) Стив МакКоннел предостерегал от использования вложенных операторов `if` — не потому что они нерациональны или неэффективны, а потому что они создают трудности для человеческого восприятия. Он сказал: «Чтобы понять код, вам нужно удерживать в памяти весь набор вложенных операторов `if`». Неудивительно, что исследования показывают, что вложенные условия — наиболее частый источник ошибок по сравнению со всеми остальными программными конструкциями. У нас на этот счет есть ряд анекдотичных случаев, которые вы сможете прочитать в разделе «Изменчивое прошлое DiffMerge».

Поэтому значение отступов для каждого уровня вложенности не в том, чтобы сделать код более понятным, а в том, чтобы служить становлению шифровальщика, лучше разбирающегося в непонятности. «Семь принципов» советуют программистам «избегать отступов», то есть писать код без глубоких вложений. «Это наиболее трудная составляющая всей этой практики, — допускает Севалд, — поскольку она требует большей изобретательности и часто может оказывать влияние на выполнение отдельных функций».

В своей книге «Code Complete», в разделе «Taming Dangerously Deep Nesting» («Снижение опасности глубокого вложения») Стив МакКоннел демонстрирует ряд полезных примеров реализации кода. В DiffMerge широко применяются два из них: операторы `case` и таблицы решений. Конечный результат, который мож-

но увидеть в исходном коде DiffMerge, заключается в том, что сам код приобретает схематическую форму, позволяющую нам распознавать крупноформатную логику, просматривая левую часть страницы сверху вниз. Наш мозг, приспособленный к чтению описаний в тексте, изложенных обычным языком, находит такую форму более простой для восприятия, чем поперечную V-образную форму глубоко вложенного кода.

Перемещение по коду

В той или иной мере в DiffMerge проиллюстрированы все Семь принципов. Например, код DiffMerge построен из отдельных логических блоков. Каждый блок выполняет отдельную задачу или какую-то разновидность задачи, и все, что он делает, либо очевидно само по себе, либо описано в предшествующем блоку комментарии. Такой код является результатом совета, взятого из «Семи принципов» о высвобождении и комментировании блоков кода. Это похоже на хорошо организованное описание, где списки определений и озаглавленные разделы помогают читателям перемещаться по кратко изложенной технической информации.

Отсутствие лишних элементов, которые только загромождают код, также облегчает перемещение по тексту DiffMerge. Один из приемов, примененных в DiffMerge для избавления от всего лишнего, заключается в использовании очень коротких имен переменных, на которые в коде присутствуют неоднократные ссылки. Это, разумеется, противоречит мудрому совету использовать значимые и описательные имена переменных. Но суть в том, что злоупотребление длинными именами настолько загромождает код, что это лишь затрудняет его понимание. Писатели это хорошо знают и именно поэтому используют в своей прозе местоимения, фамилии и сокращения.

Комментарии в DiffMerge также приведены в порядок. В коде, имеющем десятилетнюю историю, нетрудно было оставить такое количество комментариев, в котором описывается лишь то, как код используется для работы, наряду с дополнительными комментариями о том, что претерпело изменения, в виде пометок, описывающих нынешний код. Но нет никакого смысла держать в самом коде описание истории развития программы; ваша система управления исходным кодом обладает всей этой информацией и предлагает куда более подходящие способы для ее отслеживания. (Примеры этого будут приведены в следующем разделе.) Программисты, занимающиеся DiffMerge, проделали хорошую работу по, так сказать, поддержанию общественных мест в чистоте. То же самое распространяется и на сам код. В DiffMerge старый код просто не комментируют, это уже *пройденный этап*. Весь остальной код и комментарии не загромождены историческими экскурсами¹.

В коде DiffMerge допускается также свободное использование пустых пространств. Вдобавок к сокращению всяческих нагромождений пустые пространства

¹ Один из комментариев, описывающих изменение: «2-18-97 (seiwald) – translated to C++». Этот комментарий оставлен в коде как исторический раритет.

делают текст понятнее. Когда с помощью пустых пространств наподобие книжного текста выделяются и похожие образы, они приобретают визуальные очертания, распознаваемые нашим мозгом. Когда мы просматриваем код, наш мозг отмечает эти образы; позже мы уже подсознательно используем их для поиска кода, который запомнили при чтении.

Даже после того как он годами претерпевал многочисленные изменения, вносимые многими различными программистами, код DiffMerge остается во многом последовательным в своем визуальном восприятии; каждый, кто работал над DiffMerge, приложил усилия к «общей гармонии». То есть каждый из этих людей подчинил свой собственный стиль и предпочтения общей цели сделать внешний вид своего кода DiffMerge похожим на весь остальной код этой программы. Сочетаемость привела к последовательности, сократившей необходимый нашему мозгу объем работы. Это усилило результативность всех приемов удобочитаемости, которые мы только что рассмотрели.

Если вы зайдете на веб-сайт <http://www.perforce.com/beautifulcode>, то заметите что код DiffMerge не является совершенством даже по стандартам «Семи принципов». В нем есть еще места, которые могли бы быть более похожими на книгу или, к примеру, где код мог бы удачнее сочетаться со всем остальным. Конечно, мы хотели бы все это подчистить, но наряду с тем что нам нравится красивый код, еще больше нам нравится цельность объединения. Изменение имен переменных, пустых пространств, разрывов строк и т. д. может стать куда большим препятствием для объединения, чем логические изменения. Когда подобные изменения проводятся в одной из ветвей, мы увеличиваем риск, что проведенные совместно с другой ветвью устраниния ошибок приведут к *созданию* новых ошибок. Все преимущества, которые могут быть получены от переписывания неприглядных частей DiffMerge, должны быть сопоставлены с теми затратами, которые понадобятся для восстановления испортившегося объединения частей. В разделе «Изменчивое прошлое DiffMerge» мы расскажем, что происходит, когда чаша весов склоняется именно к этому.

Используемый нами инструментарий

Когда мы занимаемся чтением исходного файла, нам, естественно, нужно, чтобы код был понятен. Нам также нужно понимать код, когда мы сталкиваемся со связанными с ним изменениями, объединениями, правками, отладками, проверками, сообщениями компилятора, во многих других ситуациях и при использовании различных инструментальных средств. Оказывается, код, написанный в соответствии с «Семью принципами», читается намного легче в большинстве инструментальных средств, используемых для работы с ним.

К примеру, код DiffMerge вполне читается даже без синтаксической подсветки. Иными словами, для его чтения необязательно применять редакторы, воспринимающие контекст исходного кода. Он практически также хорошо читаем и при отображении в виде простого текста в отладчиках, компиляторах и веб-браузерах. Вот как выглядит фрагмент DiffMerge в gdb:

```

Breakpoint 3. DiffMerge::DiffDiff (this=0x00e10c0) at diff/diffmerge.cc:510
510      Mode initialMode = d.mode;
(gdb) list 505.515
505      DiffGrid d = diffGrid
506          [ df1->StartLeg() == 0 ][ df1->StartBase( ) == 0 ]
507          [ df2->StartLeg() == 0 ][ df2->StartBase( ) == 0 ]
508          [ df3->StartL1( ) == 0 ][ df3->StartL2( ) == 0 ];
509
510      Mode initialMode = d.mode;
511
512          // Pre-process rules, typically the outer snake information
513          // contradicts the inner snake. its not perfect, but we use
514          // the length of the snake to determine the best outcome.
515
(gdb) print d
$1 = {mode = CONF, diffs = DD_CONF}
(gdb)

```

Когда ведется работа с кодом, который изменяется столь же часто как Diff-Merge (он с момента первоначального написания подвергался изменениям 175 раз), программисты проводят значительную часть времени, рассматривая его в инструментальных средствах по обнаружению отличий и объединений. Общей чертой этих инструментальных средств является ограниченный горизонтальный обзор исходных файлов и привнесение собственного беспорядка. Но код наподобие DiffMerge легко читается даже в этих условиях. В просмотрщиках изменений, работающих в командной строке, его строки помещаются целиком, без переноса на следующую строку. В графических просмотрщиках изменений, чтобы увидеть его строки целиком, нам не нужно гонять из стороны в сторону горизонтальную прокрутку (рис. 32.1¹).

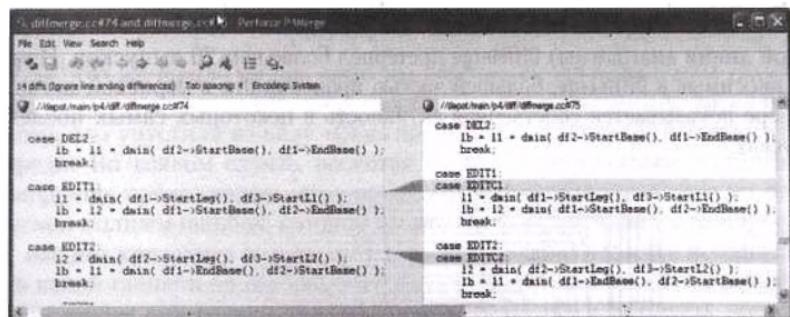


Рис. 32.1. Код DiffMerge, просматриваемый в графическом инструментальном средстве, отображающем различия

А как показано на рис. 32.2, код DiffMerge, имеющий книгообразный вид, чувствует себя довольно просторно в активно использующем боковое пространство аннотированном просмотрщике истории.

¹ Это копия экрана P4Merge, графического инструмента на основе самого DiffMerge.

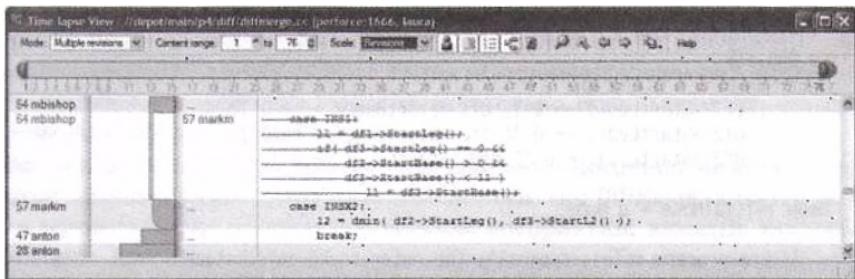


Рис. 32.2. Код DiffMerge в аннотированном просмотрщике истории

Применение книгообразного формата не только облегчает чтение кода в инструментальных средствах по объединению блоков кода, но и делает сам код более удобным для объединения. С одной стороны, полем редактирования проще управлять, когда логические блоки выделены свободным пространством. А с другой — меньшее использование вложенного кода подразумевает пропорциональное уменьшение случаев с переставленными и перепрыгивающими разделителями блоков, с которыми нужно разбираться отдельно.

Изменчивое прошлое DiffMerge

У нас есть отчет о каждом изменении, ответвлении и объединении, затрагивающем DiffMerge на всем протяжении его десятилетней истории. Это весьма любопытный документ. На рис. 32.3 предложен краткий обзор изменений выпущенных версий DiffMerge. На нем показано, что рожденный в корневой ветви (самой низкой линии диаграммы) DiffMerge претерпел более чем 20 выпусков. Изменения, вносимые в DiffMerge, большей частью происходили в корневой ветви. Но в обзоре показывается собственная активность в некоторых самых последних выпусках.

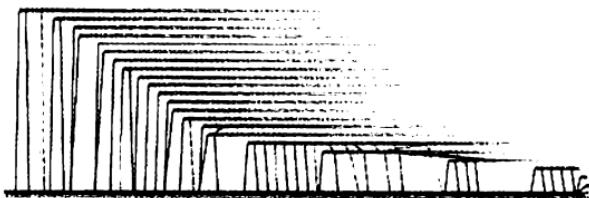


Рис. 32.3. Ветви выпусков DiffMerge

Еще более интересным представляется количество исправлений DiffMerge, приходящихся на каждый выпуск, отображенное на рис. 32.4. Там показано, что DiffMerge после своего первого выпуска редко подвергался исправлениям вплоть до выпуска 2004.2. Затем степень исправлений после этого выпуска воз-

растает и снижается снова в выпуске 2006.2. Почему же выпуски с 2004.2 по 2006.1 столь богаты на исправления?

Предыстория здесь следующая: DiffMerge начал свой жизненный путь в качестве вполне работоспособной, но довольно простой программы. На ранней стадии своей жизни он слабо различал действительные противоречия объединения и непротиворечивые изменения в соседних строках. В 2004 году мы усовершенствовали DiffMerge, сделав его более разборчивым в отношении определения и разрешения противоречий. Начиная с выпуска 2004.2 возможности DiffMerge, конечно же, возросли, но он начал «глючить». Мы получили множество сообщений об ошибках в выпусках 2004.2 и 2005.1, соответственно, было сделано большое количество исправлений. Мы попытались очистить код для выпуска 2005.2, но очистка привела к столь серьезной ошибке, что мы вынуждены были восстановить версию 2005.1 и превратить ее в выпуск 2005.2. Затем, в 2006 году, мы полностью переписали проблемные участки кода DiffMerge. Эта переделка оказалась довольно успешной, хотя и потребовала небольших поправок в выпуске 2006.1. С тех пор DiffMerge проявлял отмеченную стабильность, и степень его исправлений после выпуска снизилась до нуля.

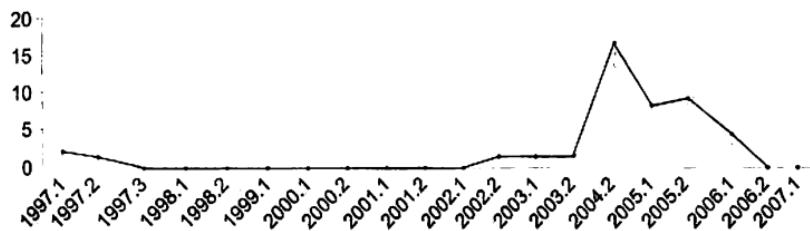


Рис. 32.4. Количество исправлений, которым подвергался DiffMerge от выпуска к выпуску

Итак, что же пошло не так, когда мы переписали DiffMerge в 2004 году? Мы полагаем, что допустили неразборчивость кода. Возможно, что во время его пересмотра мы упустили из виду «Семь принципов» или кое-что вообще не пересмотрели. Во всяком случае, несмотря на то что регressiveное тестирование DiffMerge еще продолжалось, этот компонент былпущен в плаванье по ветвям выпусков полным ошибок, которые мы не заметили, поднимаясь на борт.

У нас нет критериев оценки того, насколько хорошо читается исходный код или в какой степени он сообразуется с «Семью принципами». Но в ретроспективе мы видим подсказку, которая обязательно дала бы о себе знать, если бы мы ее в тот момент. На рис. 32.5 показан подсчет операторов `if` и их последующих вложенных уровней (сделанный на основе глубины их отступов) в каждой исходной ветви DiffMerge. К тому времени, когда была выпущена ветвь DiffMerge для 2004.2, совершенно очевидно, что мы удвоили в коде количество операторов `if`. И впервые появились операторы `if`, количество уровней вложений в которых было больше трех.

Соотношение, как говорится, не является причинной обусловленностью, и вполне возможно, что были и другие содействующие факторы. Конструкция расширений, совокупность тестовых данных, другие программные построения,

даже размер файла исходного кода — любая составляющая или все они в совокупности могли содействовать повышению уровня ошибок. Но учитывая то, что нам известно о глубоко вложенных условных выражениях и степени понятности кода, этому весьма очевидному соотношению довольно трудно не отдать должное.

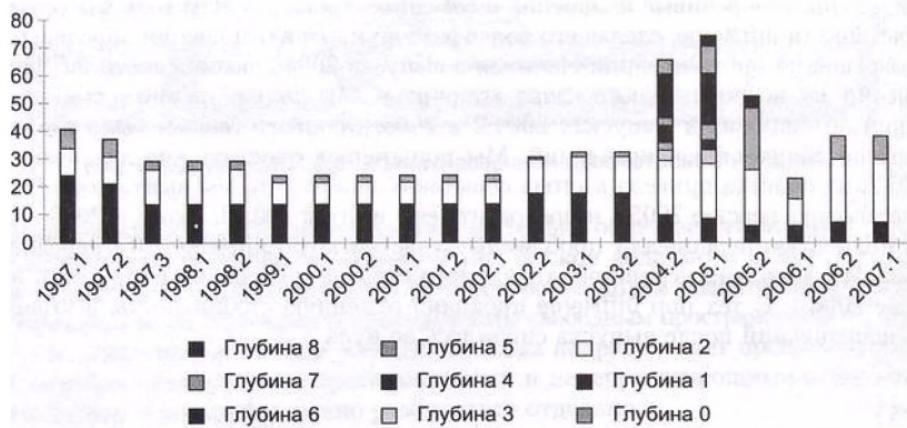


Рис. 32.5. Количество операторов if в DiffMerge, выраженное также в глубине последовательных отступов, приходящихся на каждый выпуск

Реконструкция DiffMerge 2006 года велась с прицелом на избавление от отступов. В процессе реконструкции были удалены глубоко вложенные условные выражения, а также операторы switch, чьи значения case фигурировали в новой таблице решений diffGrid. Таблица, формат которой был разработан с учетом удобства чтения, содержала список всех текущих обрабатываемых условий и предоставляла место для тех условий, которые со временем потребовалось бы обрабатывать. Таким образом, мы не только заменили проблемный код, но и подготовили почву для будущих расширений.

Вывод

Для программиста, чья разработка находится в постоянном развитии, красивым считается такой код, который может быть изменен с минимальной ценой. Вы читаете код, определяете, что он делает, и производите изменения. Успех всего дела во многом зависит от того, насколько хорошо вы поняли код, чтобы приступить к работе, поскольку этим закладывается основа для возможности программирования. Он также зависит от того, насколько хорошо ваш код понят следующим программистом, который перехватывает эстафету, и если он никогда не звал вас на помощь, значит, все сделано хорошо.

Если бы мы должны были сократить повествование этой главы, то сказали бы только о том, что успех кода в развитии зависит от того, насколько он понятен читающим его программистам. Но это ни для кого не является новостью.

Новым является то, что программисты читают код в различиях, исправлениях, объединениях, ошибках компилятора и отладчиках, а не только в редакторах с подсветкой синтаксиса, и что они довольно часто, пусть и подсознательно, извлекают логическое построение из визуального образа, в дополнение к информации, извлекаемой из самого кода. Иными словами, код понятнее, когда он привлекает внимание.

В этой главе мы исследовали эффект использования «Семи основополагающих принципов написания красивого кода» в качестве руководства для повышения понятности кода в разнообразных контекстах. Семь принципов помогли нам добиться успеха. Мы воспользовались ими для написания кода, который может развиваться с потоком вносимых изменений, и мы полагаем, что это красиво.

Благодарности

В развитие DiffMerge свой вклад внесли Кристофер Сейвальд (Christopher Seiwald), Джеймс Стриклэнд (James Strickland), Джефф Энтон (Jeff Anton), Марк Миарз (Mark Mears), Каэдмон Айриез (Caedmon Irias), Лэй Бразингтон (Leigh Brasington) и Майкл Бишоп (Michael Bishop). Авторские права на исходный код DiffMerge принадлежат Perforce Software.

Дополнительная информация

- *Kim, S.*, «Adaptive Bug Prediction by Analyzing Project History», Ph. D. Dissertation, Department of Computer Science, University of California Santa Cruz, 2006.
- *McConnell, S.*, «Code Complete», Microsoft Press, 1993.
- *McMullin, J., Varnhagen, C. K., Heng, P., and Apedoe, X.*, «Effects of Surrounding Information and Line Length on Text Comprehension from the Web», *Canadian Journal of Learning and Technology*, Vol. 28, No. 1, Winter/hiver, 2002.
- *O'Brien, M. P.*, «Software Comprehension – A Review and Direction», Department of Computer Science and Information Systems, University of Limerick, Ireland, 2003.
- *Pan, K.*, «Using Evolution Patterns to Find Duplicated Bugs», Ph.D. Dissertation, Department of Computer Science, University of California Santa Cruz, 2006.
- *Reichle, E. D., Rayner, K. и Pollatsek, A.*, «The E-Z Reader Model of Eye Movement Control» в «Reading: Comparisons to Other Models», Behavioral and Brain Sciences, Vol. 26, No. 4, 2003.
- *Seiwald, C.*, «The Seven Pillars of Pretty Code», Perforce Software, 2005, <http://www.perforce.com/perforce/papers/prettycode.html>.
- *Tufte, Edward R.*, «The Cognitive Style of PowerPoint», Graphics Press LLC, 2004.
- *Whitehead, J., and Kim, S.*, «Predicting Bugs in Code Changes», Google Tech Talks, 2006.

33 Создание программ для «Книги»

Брайан Хэйес (*Brian Hayes*)

Математик Пал Эрдьёш (Paul Erdős) часто говорил о *Книге*, легендарном фолианте (который невозможно найти на полках земных библиотек), содержащем записи наилучших доказательств всех математических теорем. Возможно, есть также *Книга* для программ и алгоритмов, в которой перечислены лучшие решения каждой вычислительной проблемы. Чтобы удостоиться своего места на ее страницах, программа должна быть не просто правильной, но также и ясной, изящной, краткой и даже остроумной.

Все мы стремимся создавать подобные драгоценные камни алгоритмического мастерства. И все мы время от времени сражаемся с упрямым битом кода, который, как его не полируй, никак не хочет сиять бриллиантом. Даже если программа выдает правильные результаты, в ней все равно остаются какие-то неуклюжие фрагменты. Логика представляет собой переплетение частных случаев и исключений к исключениям, а вся структура кажется хрупкой и ненадежной. Затем неожиданно вы испытываете прилив вдохновения или, может быть, приятель этажом ниже показывает вам какой-нибудь новый трюк, и внезапно получается то, что вполне достойно попасть в *Книгу*.

В этой главе рассказывается история одного из подобных сражений. У этого рассказа счастливое завершение, но решение о том, достойна ли получившаяся в итоге программа помещения на страницы *Книги*, я оставляю за читателями. Не стану кривить душой, но это один из тех случаев, когда озарение исходило не от меня, а от того самого приятеля этажом ниже, или, скорее, от приятеля, который находился на другом континенте.

Нелегкий путь

Речь пойдет о программе из области вычислительной геометрии, которая, как представляется, особенно богата на первый взгляд простыми проблемами, при

проникновении в суть которых становящимися настоящими головоломками. Что я вкладываю в понятие вычислительной геометрии?

Это не то же самое, что и компьютерная графика, хотя они тесно связаны друг с другом. Алгоритмы вычислительной геометрии живут не в мире пикселов, а в том идеализированном царстве линий и окружностей, где точки не имеют размеров, линии обладают нулевой толщиной, а окружности идеально круглые. В этих алгоритмах получение точных значений играет весьма существенную роль, поскольку даже малейшая погрешность может сильно изменить результат вычислений. Изменение какой-нибудь цифры, отстоящей от десятичной точки далеко вправо, способно вывернуть мир наизнанку.

Есть предположение, что Эвклид как-то сказал своему ученику из семьи правителей, что «к геометрии не проторено королевских путей». Среди некоролевских путей дорога к вычислениям является особенно запутанной, неровной и ухабистой. Иногда сопутствующие трудности имеют отношение к вычислительной эффективности: время работы программы и расход оперативной памяти должны удерживаться в рамках разумного. Но в данном случае основной интересующей меня проблемой геометрических алгоритмов является не эффективность, а концептуальность и эстетичность решения поставленных передо мной довольно непростых задач. Сможем ли мы с ними справиться? И сможем ли мы сделать это красиво?

Представленная далее в нескольких вариантах программа должна ответить на элементарный вопрос: если на плоскости заданы три точки, то лежат ли они на одной и той же прямой? Это звучит настолько просто, что и решение должно быть несложным. Несколько месяцев назад, когда мне понадобилась в составе одной большой программы подпрограмма для ответа на вопрос о коллинеарности (размещения на одной прямой), задача казалась настолько простой и понятной, что я даже не удосужился покопаться в литературе и посмотреть на ее возможные решения. Я не сожалею о своей поспешности — попытка самостоятельного решения проблемы кое-чему меня научила или по крайней мере создала хорошую репутацию — но я должен признать, что это была отнюдь не королевская дорога. В итоге я повторил шаги своих предшественников. (Может быть, из-за этого дорога так сильно разбита, и имеет не одну и не две колеи!)

Предупреждение для тех, кто боится скобок

Код представлен на языке Lisp. Я не собираюсь извиняться за свой выбор языка программирования, но при этом и не хочу превращать эту главу в трактат для рекрутования приверженцев Lisp. Хочу лишь заметить, что верю в пользу многоязычности. Если при чтении представленных далее фрагментов кода вы узнаете что-нибудь новое о незнакомом вам языке, то это вряд ли причинит вам какой-нибудь вред. Все процедуры очень короткие, с полдюжины строк. На рис. 33.1 представлено краткое руководство по структуре Lisp-процедуры.

Кстати, алгоритм, осуществляемый программой, представленной на рисунке, несомненно, взят из *Книги*. Это алгоритм Эвклида для вычисления наибольшего общего делителя двух чисел.



Рис. 33.1. Составные части определения Lisp-процедуры

Три в ряд

Если бы вы решали проблему коллинеарности с карандашом и бумагой, то как бы вы поступили? Было бы вполне естественным нарисовать положения трех точек на миллиметровке, а затем, если ответ на первый взгляд не был бы очевиден, провести прямую между двумя точками и посмотреть, не проходит ли она и через третью точку (см. рис. 33.2). Если она проходит в непосредственной близости, то точность в нанесении точек и вычерчивании прямой становится насущной потребностью.

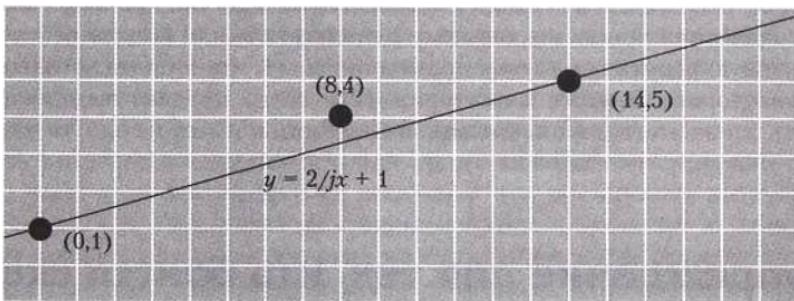


Рис. 33.2. Три неколлинеарные точки

Компьютерная программа может сделать то же самое, хотя для компьютера не может быть ничего «очевидного при рассмотрении». Чтобы провести прямую через две точки, программа получает уравнение этой прямой. Чтобы определить, находится ли третья точка на этой прямой, программа проверяет, удовлетворяют или нет координаты этой точки данному уравнению. (Упражнение:

для произвольного набора трех заданных точек есть три пары точек, которые можно выбрать для соединения прямой линией, оставляя в каждом случае другую, третью точку для проверки на коллинеарность. Для некоторых выбранных вариантов задача может решаться проще, чем для других, в том смысле, что для этого нужна меньшая точность. Существуют ли какие-нибудь простые критерии для принятия решения?)

Уравнение прямой принимает вид $y = mx + b$, где m — это наклон, а b — это y -пересечение, точка (если таковая существует) пересечения с координатой y . Итак, при заданных трех точках p , q и r нужно найти значения m и b для прямой, которая проходит через две из них, а затем проверить x - и y -координаты третьей точки на соответствие тому же уравнению.

Вот как выглядит программный код:

```
(defun naive-collinear (px py qx qy rx gy)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (= gy (+ (* m rx) b))))
```

Процедура является утверждением: она возвращает булево значение `true` или `false` (на жаргоне Lisp `t` или `nil`). Шесть аргументов представляют собой x - и y -координаты точек p , q и r . Выражение `let` вводит локальные переменные m и b , связывая их со значениями, возвращаемыми процедурами вычисления наклона — `slope` и пересечения — `y-intercept`. Я еще вернусь к краткому определению этих процедур, но что они делают, можно понять по их названию. И наконец, в последней строке процедуры делается вся работа по ответу на поставленный вопрос: равна ли y -координата точки r x -координате точки r , умноженной на m , плюс b ? Ответ возвращается в виде значения функции `naive-collinear`.

Можно ли сделать все проще? Посмотрим. Работоспособна ли эта программа? Обычно да. Если использовать процедуру с большим набором случайно сгенерированных точек, то скорее всего она будет работать очень долго и безотказно. Тем не менее ее легко сбить с толку. Стоит лишь применить ее к точкам $(x\ y)$ с координатами $(0\ 0)$, $(0\ 1)$ и $(0\ 2)$. Коллинеарность этих точек не вызывает сомнения — все они лежат на оси y — и все же не приходится рассчитывать на то, что процедура `naive-collinear` даст какой-либо разумительный ответ, получив эти точки в качестве аргументов.

Основная причина такого отказа кроется внутри определения функции `slope`. С точки зрения математики наклон m — это $\Delta y / \Delta x$, вычисляемый программой следующим образом:

```
(defun slope (px py qx qy)
  (/ (- qy py) (- qx px))))
```

Если получается, что p и q имеют одинаковую x -координату, тогда Δx равна нулю, и выражение $\Delta y / \Delta x$ становится неопределенным. Если вы попытаетесь вычислить наклон, то не получите ничего, кроме ошибки деления на нуль. Справиться с этой неприятностью можно множеством способов. Когда я впервые составлял фрагменты этой небольшой программы, мой выбор пал на возвращение функцией `slope` специального предупреждающего значения, если px было равно qx . Обычно в Lisp для этих целей выбирают значение `nil`:

```
(defun slope (px py qx qy)
  (if (= px qx)
      nil
      (/ (- qy py) (- qx px))))
```

Как и наклон, у-пересечение вертикальной прямой так же не определено, поскольку эта прямая либо нигде не пересекается с осью y , либо ($x=0$) совпадает с ней. Здесь применяется тот же прием с использованием `nil`:

```
(defun y-intercept (px py qx qy)
  (let ((m (slope px py qx qy)))
    (if (not m)
        nil
        (- py (* m px)))))
```

Теперь нужно переделать вызываемую процедуру, чтобы в ней учитывалась возможность получения наклона m не в виде числа, а в виде подставного значения:

```
(defun less-naive-collinear-p (px py qx qy rx ry)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (if (numberp m)
        (= ry (+ (* m rx) b))
        (= px rx))))
```

Если m имеет числовое значение — выражение (`numberp m`) возвращает `t` — все идет как и раньше. В противном случае я знаю, что p и q имеют одну и ту же x -координату. Из этого следует, что три точки коллинеарны, если r также имеет то же самое значение x .

По мере развития программы необходимость введения специальных мер предосторожности для вертикальных прямых вызывала постоянное раздражение. Получалось, что каждая написанная мной процедура должна была иметь какую-нибудь уродливую правку, «прикрученную» для того, чтобы учитывать возможность наличия прямой, параллельной оси y . Следует признать, что правка была всего лишь выражением `if` и дополнительной парой строк кода и не представляла собой важной проблемы искусства программирования. Но по своему характеру она казалась мне лишним вычислением, сигнализируя, возможно, о том, что я делаю что-то не так или неоправданно усложняю ситуацию. Вертикальные прямые в общем-то не отличаются от горизонтальных или от тех, которые лежат на плоскости под любым другим углом. Измерять наклон относительно оси y является произвольной условностью; если мы выберем другое исходное направление, мир от этого не перевернется.

Это наблюдение предлагает способ обхода проблемы: нужно развернуть всю систему координат. Если набор точек коллинеарен в одной системе, то эти же точки должны быть коллинеарны и во всех других системах. Наклоните оси на несколько градусов в ту или иную сторону, и тупиковая ситуация, связанная с делением на нуль, исчезнет. Вращение не является какой-то затруднительной или требовательной к вычислительным ресурсам задачей; это всего лишь матричное умножение. С другой стороны, такой подход означает, что мне по-прежнему нужно где-нибудь использовать выражение `if`, чтобы проверить, действительно ли px равно qx . Но я предпочел бы упростить логику и вообще изба-

виться от точки ветвления. Есть ли возможность проверить на коллинеарность, используя простые средства вычисления координат точек без проведения какого-либо анализа?

Вот что было рекомендовано (в несколько видоизмененном контексте) на одном из веб-сайтов, который я позволю себе не называть: когда Δx равна 0, нужно просто установить значение $\Delta y / \Delta x$ равным 10^{10} , что «достаточно близко к бесконечности». Из практических соображений я думаю, что такая тактика в большинстве случаев может действительно неплохо сработать. В конце концов, если входные параметры программы берутся каким-то образом из измерений реального мира, то погрешности будут куда больше, чем 1 часть от 10^{10} . И все-таки эту стратегию я не могу воспринимать всерьез. Я могу и не знать, как выглядит версия коллинеарности, занесенная в *Книгу*, но отказываюсь верить в то, что в ней есть параметры, определяемые как «достаточно близкие к бесконечности».

Скользящий наклон

Вместо проведения прямой через две точки и выяснения, лежит ли третья точка на этой же прямой, предположим, что я нарисовал все три прямые и провел, не являются ли они одной и той же прямой линией. Вообще-то для этого нужно нарисовать только две прямые, поскольку если прямая pq идентична прямой qr , то она также должна совпадать и с прямой pr . К тому же получается, что мне нужно сравнить лишь углы наклона, не привлекая к этому у-пересечения. (Вы уже поняли почему?) Оценка на глазок совпадения двух линий или формирования ими небольшого угла расхождения вряд ли могла стать самой надежной процедурой, но в мире вычислений все сводится к простому сравнению двух чисел, значений m (рис. 33.3).

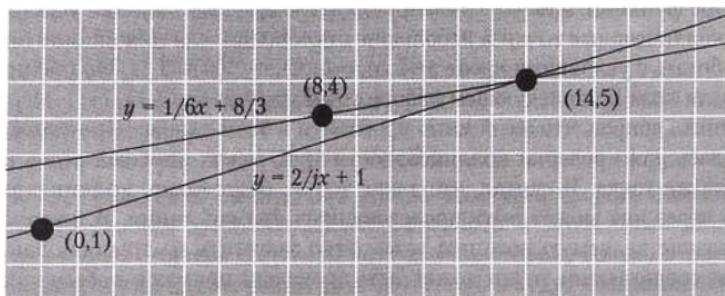


Рис. 33.3. Проверка коллинеарности путем сравнения наклонов

Я написал следующую версию функции `collinear`:

```
(defun nm-collinear (px py qx qy rx ry)
  (equalp (slope px py qx qy)
          (slope qx qy rx ry)))
```

Это уже намного лучше! И выглядит куда проще. Здесь нет выражения `if`, отвлекающего внимание на характерное состояние вертикальных прямых; все наборы точек рассматриваются одинаковым образом.

Тем не менее я вынужден признать, что простота и кажущееся единобразие – это всего лишь иллюзия, основанная на некоторых закулисно происходящих Lisp-ухищрениях. Заметьте, что я использую для сравнения наклонов не оператор `=`, а универсальную функцию возврата логического значения – `equalp`. Процедура работает правильно только потому, что `equalp`, как оказывается, верно обрабатывает возвращенное `slope` число или значение `#t`. (То есть два наклона считаются равными, если оба они характеризуются одинаковым числом, или если оба они имеют значение `#t`.) На языке с более вычурной системой типов определение уже не привлекало бы своей краткостью. Потребовалось воспользоваться чем-нибудь вроде следующего фрагмента:

```
(defun typed-mm-collinear (px py qx qy rx ry)
  (let ((pq-slope (slope px py qx qy))
        (qr-slope (slope qx qy rx ry)))
    (or (and (numberp pq-slope)
              (numberp qr-slope)
              (= pq-slope qr-slope))
        (and (not pq-slope)
              (not qr-slope)))))
```

Здесь внешний вид уже не столь привлекателен, хотя даже в этой, более конкретизированной форме, логика представляется мне менее извращенной, чем в первоначальной «примитивной» версии. Весь смысл здесь состоит в том, что `px` и `qx` являются одной и той же прямой, если оба наклона выражены числами и эти числа одинаковы или если оба наклона выражены значениями `#t`. Но стоит ли упрекать «умную» Lisp-программу только за то, что другие языки на подобный трюк не способны?

На этом я хотел было и закончить и принять `mm-collinear` в качестве окончательной версии программы, но тестирование выявило другое ненормальное явление. Обе функции, и `mm-collinear` и `less-naive-collinear`, вполне успешно различали коллинеарные точки и небольшие отклонения, для них вполне по силам было работать с условиями вроде $p=(0\ 0)$, $q=(1\ 0)$, $r=(1000000\ 1)$. Но обе процедуры терпели крах при следующем наборе точек: $p=(0\ 0)$, $q=(0\ 0)$, $r=(1\ 1)$.

Возникал вопрос, что же должно произойти в этом случае. Программа предназначалась для проверки коллинеарности трех точек, но здесь `p` и `q` по сути одна и та же точка. Я считал, что такие точки, безусловно, коллинеарны, поскольку через них может быть проведена одна прямая линия. Я допускал, что и противопоставляемую позицию тоже легко защитить на том основании, что через две совпадающие точки может быть проведена прямая с *любым* наклоном. К сожалению, обе процедуры в том виде, в каком они были написаны, не соответствовали *ни одному* из этих правил. Для приведенного выше примера они возвращали `#t` и возвращали `t` для точек $p=(0\ 0)$, $q=(0\ 0)$ и $r=(0\ 1)$. Разумеется, это с любой точки зрения было ненормально.

Можно было решить эту проблему декларативно, признав, что эти три аргумента процедуры должны быть выделены как особые точки. Но тогда нужно было написать код для отлова исключений из правил, вызова исключений, воз-

вращения значений, вызывающих ошибку, разборки с нарушителями и т. д. Всем этим совершенно не стоило заниматься.

Неравенство в треугольнике

А вот еще один способ переосмыслиния проблемы. Заметьте, что если p , q и r не коллинеарны, то они образуют треугольник (рис. 33.4). Свойство любого треугольника заключается в том, что самая длинная сторона всегда короче, чем сумма длин двух его меньших сторон. Но если три точки коллинеарны, треугольник сплющивается, и самая длинная «сторона» в точности становится равной сумме более коротких «сторон».

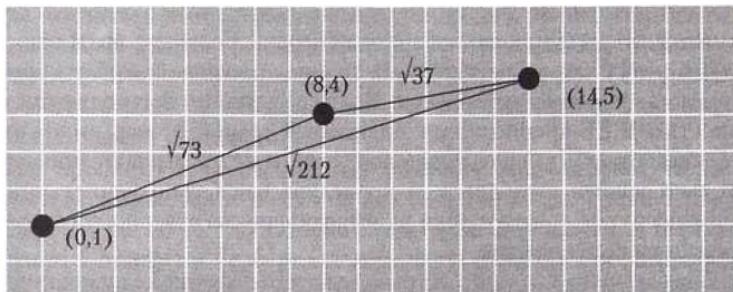


Рис. 33.4. Проверка коллинеарности за счет неравенства в треугольнике

(Для примера, показанного на этом рисунке, более длинная сторона короче, чем сумма двух других сторон, примерно на 0,067.)

Код этой версии функции уже не столь лаконичен, но все, что в нем происходит, смотрится достаточно просто:

```
(defun triangle-collinear (px py qx qy rx ry)
  (let ((pq (distance px py qx qy))
        (pr (distance px py rx ry))
        (qr (distance qx qy rx ry)))
    (let ((sidelist (sort (list pq pr qr) #'>)))
      (= (first sidelist)
         (+ (second sidelist) (third sidelist))))))
```

Идея заключается в вычислении длины всех трех сторон, помещения их в список, сортировке этого списка по убывающей, а затем сравнении первой (самой длинной) стороны с суммой двух других сторон. Только в том случае, если эти показатели длины равны друг другу, точки будут коллинеарны. Достоинства данного подхода дают повод для его рекомендации. Вычисление зависит только от геометрических отношений между самими точками; оно не зависит от их местоположения и ориентации на плоскости. Наклоны и пересечения даже не упоминаются. К тому же в качестве дополнительного преимущества эта версия процедуры дает последовательные и разумные ответы, когда две или три точки совпадают друг с другом: все подобные наборы точек считаются коллинеарными.

К сожалению, за эту простоту приходится дорого расплачиваться. До настоящего момента все вычисления выполнялись чисто арифметически. Если исходные координаты определены в значениях целых или рациональных чисел, то наклоны и пересечения вычислялись без погрешностей округлений или иных ошибок. Например, прямая, проходящая через точки (1 1) и (4 2), имеет наклон $m=1/3$ и у-пересечение $b=2/3$ (без десятичных приближений вроде 0,33 и 0,67). Если числа представлены таким образом, сравнение гарантирует получение математически верного ответа. Но точность при измерении расстояний недостижима. Процедура `distance`, вызываемая из `triangle-collinear`, определена следующим образом:

```
(defun distance (px py qx qy)
  (sqrt (+ (square (- qx px))
            (square (- qy py)))))
```

Разумеется, во всем виноват квадратный корень. Если `sqrt` возвращает иррациональный результат, то надеяться на точное, имеющее какой-то предел числовое представление не приходится. Когда расстояния вычисляются с двойной точностью IEEE-арифметики с плавающей точкой, `triangle-collinear` выдает надежные ответы для точек, чьи координаты не превышают примерно 10^5 . Стоит только пройти этот порог, и процедура неизбежно начнет принимать очень приплюснутые треугольники за выродившиеся, неверно сообщая о том, что их вершины коллинеарны.

Способов быстрого и легкого устранения этого дефекта не существует. Трюки наподобие вращения или изменения масштабов системы координат здесь уже не помогут. Это недостаток (или свойство?) нашего мира: рациональные точки могут дать начало иррациональным расстояниям. При таких условиях получение точных и надежных результатов не то чтобы невозможно, но требует достижения определенных технических пределов. Когда три точки действительно коллинеарны, этот факт может быть доказан алгебраически, без вычисления квадратных корней. К примеру, возьмем коллинеарные точки (0 0), (3 3) и (5 5), для них уравнение расстояний имеет вид $\sqrt{50} = \sqrt{18} + \sqrt{8}$ и может быть приведено к виду $5 \times \sqrt{2} = 3 \times \sqrt{2} + 2 \times \sqrt{2}$. Когда точки *не* коллинеарны, вычисление в конечном итоге, если вы вычислите подкоренные выражения с достаточным количеством цифр, приведет к неравенству. Но я не вынашиваю идею применения аналитической алгебраической системы и адаптивно регулируемого арифметического модуля с несколькими степенями точности вычислений только для того, чтобы проверить три точки на коллинеарность. Наверное, должно быть какое-нибудь более легкое решение. Готов предположить, что в версии, помещенной в *Книгу*, применен более экономный способ.

Блуждания по извилистостям

Чтобы завершить историю, мне нужно упомянуть о той обстановке, в которой все это происходило. Несколько месяцев назад я играл с простой моделью извилистости речного русла — с формированием тех гигантских подковообраз-

ных изгибов, которые вы можете увидеть на примере нижнего течения Миссисипи. Модель разбивала гладкие изгибы русла реки на цепочку коротких, прямолинейных отрезков. Мне нужно было измерить извилистость реки на языке углов изгиба между этими отрезками, и в частности, я хотел обнаружить области нулевого искривления — а следовательно, области, отвечающие условиям коллинеарности.

Другая часть программы принесла мне еще больше неприятностей. Поскольку изгибы увеличиваются и перемещаются, одна петля иногда набегает на другую, русло в этом месте замыкается накоротко и оставляет позади себя брошенное рогообразное озеро. (Не хотелось бы оказаться в этом месте Миссисипи именно в тот момент, когда все это происходит.) Чтобы обнаружить такие события на модели, я должен был просмотреть пересечения отрезков. И опять я мог бы посчитать подпрограмму работоспособной, но она казалась мне через чур сложной со своим деревом решений, имевшим дюжину ответвлений. Как и при вычислении коллинеарности, вертикальные отрезки и совпадающие точки требовали особой обработки, к тому же мне еще приходилось справляться с параллельными отрезками.

Решая проблему пересечений, я в конечном итоге провел часть времени в библиотеке и покопался в сети. И многому научился. Именно там я нашел подсказку, что число 10^{10} достаточно близко к бесконечности. А Бернард Казелле (Bernard Chazelle) и Герберт Эделсброннер (Herbert Edelsbrunner) предложили более изощренный способ обхода всех особенностей и вырождений, с которыми я столкнулся. В обзорной статье 1992 года, посвященной алгоритмам пересечения прямолинейных отрезков (см. раздел «Дополнительная информация» в конце этой главы), они написали:

Чтобы упростить объяснение, мы должны допустить, что никакие две конечные точки не имеют одинаковых x - или y -координат. В частности, это применимо к двум конечным точкам одного и того же отрезка и исключает, таким образом, присутствие вертикальных или горизонтальных отрезков... Наше логическое обоснование заключается в том, что ключевые идеи алгоритма лучше всего объясняются без оглядки всякий раз на особые случаи. Применять снижение строгости допущений очень удобно (не требуется никаких новых идей), но скучно. Все это хорошо в теории. Но реализация алгоритма, позволяющая программе работать в любых случаях, это весьма отпугивающая задача. Существуют также числовые проблемы, которые сами по себе достойны написания отдельной статьи. Но следуя древней традиции, мы попытаемся не слишком обо всем этом беспокоиться.

Возможно, самый важный урок, извлеченный в результате этого вторжения в литературу, заключался в том, что другие также столкнулись в этой области с весьма существенными проблемами. И дело здесь не в том, что я такой злокомплексованный программист. Это открытие меня успокоило; с другой стороны, оно ничего не дало для реального решения моей проблемы.

Чуть позже я опубликовал на своем блоге <http://bit-player.org> статью, посвященную алгоритмам пересечения прямолинейных отрезков. По сути это была просьба о помощи, и она себя ждать не заставила — помочи было столько, что

я не успевал со всем разобраться. Один читатель в качестве средства от неопределенных наклонов предложил применить полярные координаты, а другой настаивал на том, что линейные уравнения нужно переписать в параметрическом виде, чтобы координаты x и y задавались в виде функций новой переменной t . Барри Кипра (Barry Cipra) предложил несколько иную параметрическую схему, а затем придумал еще один алгоритм, основанный на идее применения афинного преобразования для смещения одного из отрезков на интервал $(-1, 0)$, $(1, 0)$. Дэвид Эппштейн (David Eppstein) настаивал на том, что бы убрать проблему из раздела Евклидовой геометрии и решать ее в проекции на плоскость, где наличие «точки в бесконечности» помогало разобраться со специфическими случаями. И наконец, Джонатан Ричард Шевчук (Jonathan Richard Shewchuk) дал мне указатель на записи его лекций, статьи и работоспособный код; далее я еще вернулся к идеи Шевчука.

Я был поражен и немного смущен этим наплывом разумных и творческих предложений. В них встретилось несколько вполне жизнеспособных кандидатов на процедуру пересечения отрезков. Более того, я нашел также ответ на проблему коллинеарности. И я действительно верю, что переданное мне решение вполне достойно стать алгоритмом, внесенным в *Книгу*.

«Что вы говорите!» — то есть «Ага!»

В комиксах момент озарения изображается вспыхнувшей лампочкой в овальной выноске, обозначающей мысль. У меня внезапная вспышка озарения больше ассоциируется с легким подзатыльником. Бывает так, что вы проснетесь после того, как что-то изучили, и ваше новое проникновение в суть вещей столь ослепительно очевидно, что вы не можете даже поверить в то, что раньше этого не знали. Еще через несколько дней вы начинаете подозревать, что, может быть, действительно знали это, вы *должно быть* знали это, но только нуждались в напоминании.

И когда вы делитесь этим открытием со следующим человеком, то начинаете со слов: «Общеизвестно, что...»

Именно такой была моя реакция на прочтение заметок к лекции по корректности в геометрии Джонатана Шевчука (Jonathan Shewchuk) «*Lecture Notes on Geometric Robustness*». Он показал алгоритм коллинеарности, который, как только я с ним разобрался, представился мне настолько естественным и разумным, что я был уверен, что все это раньше дремало где-то во мне самом. Ключевая идея заключалась в работе с площадью треугольника, а не с периметром, как в функции `triangle-collinear`. Очевидно, что площадь треугольника равна нулю лишь в том случае, если он вырожденный и его вершины коллинеарны. Но если оценивать функцию площади по сравнению с функцией периметра, то у первой есть два существенных преимущества. Во-первых, она может быть реализована без извлечения квадратных корней или других каких-нибудь операций, которые могли бы вывести нас из области рациональных чисел. Во-вторых, она значительно менее зависима от точности представления чисел.

Рассмотрим совокупность равнобедренных треугольников с вершинами $(0, 0)$, $(x, 1)$ и $(2x, 0)$. По мере роста значения x разница между длиной основания и суммой длин двух сторон постоянно уменьшается, соответственно, становится трудно отличить этот сильно уменьшенный треугольник от полностью сглаженного треугольника с вершинами $(0, 0)$, $(x, 0)$ и $(2x, 0)$. Вычисление площади не страдает от этой проблемы. Напротив, площадь постоянно растет, поскольку треугольник становится более вытянутым (рис. 33.5). В числовом отношении, даже без точной арифметики, вычисление становится намного более надежным.

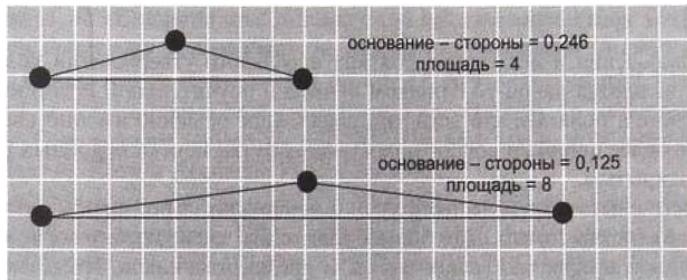


Рис. 33.5. Проверка коллинеарности за счет вычисления площади

Как вычислить площадь? Эвклидова формула $1/2bh$ не самый лучший ответ на этот вопрос, как, впрочем, и тригонометрический подход. Значительно лучше рассматривать стороны треугольника в качестве векторов. Два вектора, исходящие из любой вершины, образуют параллелограмм, площадь которого получается за счет взаимного перемножения векторов. Площадь треугольника составляет половину площади параллелограмма. Фактически вычисление дает «площадь со знаком»: результат положительный, если вершины треугольника берутся против часовой стрелки, и отрицательный, если они берутся по часовой стрелке. Но для поставленной цели более важно, что эта площадь равна нулю только в том случае, если вершины коллинеарны.

Векторная формула для вычисления площади наиболее кратко выражается в терминах определителя матрицы два на два:

$$A = \frac{1}{2} \begin{vmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{vmatrix} = \frac{1}{2} [(x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3)]$$

Поскольку меня интересует только случай, когда определитель равен нулю, я могу проигнорировать влияние значения $1/2$ и запрограммировать утверждение коллинеарности в следующей упрощенной форме:

```
(defun area-collinear (px py qx qy rx ry)
  (= (* (- px rx) (- qy ry))
      (* (- qx rx) (- py ry))))
```

Вот это то, что нужно: простая арифметическая функция x - и y -координат, требующая четырех вычитаний, двух умножений, утверждения равенства и ничего другого — никаких if , никаких наклонов, никаких пересечений, никаких квадратных корней, никакого риска возникновения ошибок деления на нуль. Если выполнение процедуры будет происходить именно в среде арифметики рацио-

нальных чисел, то она всегда будет выдавать точные и правильные результаты. Охарактеризовать поведение арифметики чисел с плавающей точкой более сложно, но куда сложнее сделать это в отношении версии, основанной на сравнении отрезков периметра. Шевчук предоставил хорошо настраиваемый код на языке Си, использующий по возможности числа с плавающей точкой и переключающийся по необходимости на точную арифметику.

Вывод

Мои приключения и невзгоды в поисках идеального утверждения коллинеарности не являются какой-то слишком ценной и поучительной историей. В конечном счете я полагаю, что встретил правильное решение именно своей проблемы, но более широкий вопрос о наилучших путях поиска подобных решений остался открытым.

Из моего опыта можно вынести урок о незамедлительной просьбе о помощи: ведь всегда есть кто-то, кто знает больше вашего. Вы также можете воспользоваться совокупной мудростью ваших коллег и предшественников. Иными словами, если Google может с некоторой долей вероятности отыскать требуемый алгоритм или даже исходный код, то к чему терять время на изобретение велосипеда?

По поводу этого совета у меня возникают несколько смешанные чувства. Когда инженер проектирует мост, я предполагаю, что он обладает основательными знаниями о том, как другие представители его профессии решали в прошлом сходные задачи. И все же профессиональные знания – это не просто умение найти и применить светлые идеи других людей, я хотел бы, чтобы мой проектировщик моста решил ряд проблем и своими силами.

Другой вопрос касается срока поддержки жизнеобеспечения «больной» программы. В этой главе рассматривались самые маленькие программы, поэтому ничего не стоило отказаться от них и начать все заново, как только выявлялась малейшая неприятность. Для более крупных проектов не так-то просто от чего-то отказаться. И такой поступок не всегда благородителен: вы меняете уже известные проблемы на еще невыявленные.

В конце концов возникает вопрос о позволительном уровне влияния поиска «красивого кода» на процесс программирования или разработки программного обеспечения. Математик Г. Харди (G. H. Hardy) провозгласил, что «для уродливой математики нет места в этом мире». Нужно ли придавать эстетическим принципам такой же вес и в компьютерной науке? Есть иной способ задать тот же самый вопрос: есть ли у нас хоть какая-то гарантия, что программа, достойная помещения в *Книгу*, существует для каждой четко сформулированной проблемы вычисления? Возможно, в *Книге* остается еще много пустых страниц.

Дополнительная информация

- Avnaim, F., J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. «Evaluating signs of determinants using single-precision arithmetic». *Algorithmica*, Vol. 17, pp. 111–132, 1997.

- Bentley, Jon L., and Thomas A. Ottmann. «Algorithms for reporting and counting geometric intersections». *IEEE Transactions on Computers*, Vol. C-28, pp. 643–647, 1979.
- Braden, Bart. «The surveyor's area formula». *The College Mathematics Journal*, Vol. 17, No. 4, pp. 326–337, 1986.
- Chazelle, Bernard, and Herbert Edelsbrunner. «An optimal algorithm for intersecting line segments in the plane». *Journal of the Association for Computing Machinery*, Vol. 39, pp. 1–54, 1992.
- Forrest, A. R. «Computational geometry and software engineering: Towards a geometric computing environment». In *Techniques for Computer Graphics* (edited by D. F. Rogers and R. A. Earnshaw), pp. 23–37. New York: Springer-Verlag, 1987.
- Forrest, A. R. «Computational geometry and uncertainty». In *Uncertainty in Geometric Computations* (edited by Joab Winkler and Mahesan Niranjan), pp. 69–77. Boston: Kluwer Academic Publishers, 2002.
- Fortune, Steven, and Christopher J. Van Wyk. «Efficient exact arithmetic for computational geometry». In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pp. 163–172. New York: Association for Computing Machinery, 1993.
- Guibas, Leonidas, and Jorge Stolfi. «Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams». *ACM Transactions on Graphics*, Vol. 4, No. 2, pp. 74–123, 1985.
- Hayes, Brian. «Only connect!» <http://bit-player.org/2006/only-connect>. [Публикация в блоге от 14 сентября 2006 г.]
- Hayes, Brian. «Computing science: Up a lazy river». *American Scientist*, Vol. 94, No. 6, pp. 490–494, 2006. (<http://www.americanscientist.org/AssetDetail/assetid/54078>)
- Hoffmann, Christoph M., John E. Hopcroft and Michael S. Karasick. «Towards implementing robust geometric computations». *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pp. 106–117. New York: Association for Computing Machinery, 1988.
- O'Rourke, Joseph. «Computational Geometry in C». Cambridge: Cambridge University Press, 1994.
- Preparata, Franco P., and Michael I. Shamos. «Computational Geometry: An Introduction». New York: Springer-Verlag, 1985.
- Qian, Jianbo, and Cao An Wang. «How much precision is needed to compare two sums of square roots of integers?» *Information Processing Letters*, Vol. 100, pp. 194–198, 2006.
- Shewchuk, Jonathan Richard. «Adaptive precision floating-point arithmetic and fast robust geometric predicates». *Discrete and Computational Geometry*, Vol. 18, pp. 305–363, 1997. Предварительная публикация доступна на веб-сайте <http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robustarithmetic.ps>.
- Shewchuk, Jonathan Richard. «Lecture notes on geometric robustness». (Версия от 26 октября 2006 г.) (<http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz>. См. также исходный текст кода <http://www.cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c>)

Послесловие

В книге проводится исследование границ человеческой изобретательности и мастерства в одной из областей приложения интеллектуальных усилий: в разработке компьютерных систем. В каждой главе красота раскрывается в уникальных решениях, источник которых обнаруживается в авторской способности заглядывать за границы обычных представлений, распознавать критические ситуации, оставленные другими без внимания, и обнаруживать неожиданные решения трудных задач.

Многие авторы борются с ограничениями в физическом окружении, в доступных ресурсах или в том же определении требований к ним, которые затрудняют даже представление о возможных решениях. Другие авторы вторгаются в те области, в которых уже были готовые решения, но привносят в них новый взгляд на вещи и уверенность, что можно достичь чего-то значительно лучшего.

Все авторы, представленные в этой книге, извлекли уроки из своих проектов. Но мы можем извлечь и более широкомасштабные уроки, если предпримем длительное и полное событий путешествие через всю книгу.

Сначала мы поймем, что порой срабатывают испытанные временем правила. Поэтому зачастую кто-то сталкивается с трудностями, пытаясь поддержать стандарты надежности, читаемости или применить другие принципы качественной разработки программного обеспечения. Бывает так, что в подобных ситуациях не стоит отказываться от столь многообещающих принципов. Иногда приподнявшись над проблемой и изучив ее со всех сторон, можно открыть новые грани, позволяющие выполнить требования, не жертвуя хорошей технологией.

В то же время некоторые главы подтверждают старое клише, что перед тем как ломать правила, их нужно знать. Некоторые авторы десятилетиями набирались опыта, прежде чем выбрать иной путь к решению какой-нибудь одной трудной проблемы, и этот опыт вселял в них уверенность, что ломка сложившихся правил осуществляется конструктивным образом.

В этой книге нашлось место и урокам, относящимся к междисциплинарным исследованиям. Многие авторы внедрились в новые для себя области. В подобных ситуациях праздновала победу особенно четкая форма творческого подхода и осмыслиения.

И наконец, прочитав книгу, мы узнаем, что красивые решения не вечно. Ноевые обстоятельства всегда требуют свежего взгляда. Поэтому если вы читаете эту книгу и думаете: «Я не смогу воспользоваться предлагаемыми решениями

в каком-либо из своих проектов», не стоит переживать, при работе над своими новыми проектами наши авторы тоже воспользуются иными решениями.

Я кропотливо работал над этой книгой около двух месяцев, помогая авторам довести затрагиваемые ими темы до совершенства и поднять выразительность основных моментов. Погружение в работу исключительно талантливых изобретателей оказало на меня воодушевляющее и даже вдохновляющее на подвиги воздействие. Я получил импульс для новых экспериментов и надеюсь, что то же самое предстоит испытать читателям этой книги.

Энди Орам (*Andy Oram*)

О тех, кто работал над книгой

Джон Бентли (Jon Bentley) — специалист по компьютерным системам в исследовательской лаборатории *Avaya*. В круг его исследовательских интересов входят технологии программирования, алгоритмы проектирования и конструирование программного инструментария и интерфейсов. Он написал ряд книг по программированию и статей на разные темы, от теории алгоритмов до разработки программного обеспечения. Он получил степень бакалавра наук в Стэндфорде в 1974 году и магистра и кандидата наук в Университете Северной Каролины в 1976 году, а затем в течение шести лет преподавал теорию вычислительных машин и систем в Университете Карнеги Меллон. В 1982 году он присоединился к персоналу исследовательской лаборатории *Bell Labs Research*, откуда ушел в 2001 году, чтобы перейти в *Avaya*. Он был нештатным преподавателем в Вест-Пойнте и Принстоне и участвовал в работе компаний, поставляющих инструментальные средства для разработки программного обеспечения, телефонные коммутаторы, телефоны и веб-службы.

Тим Брэй (Tim Bray) участвовал в управлении проектом создания Оксфордского словаря — *Oxford English Dictionary* в Университете Ватерлоо в Онтарио, Канада, в 1987–1989 годах, был соучредителем *Open Text Corporation* в 1989 году, участвовал в запуске одного из первых общедоступных движков веб-поиска в 1995 году, был одним из создателей XML 1.0 и соредактором «Namespaces in XML» между 1996 и 1999 годами, основателем *Antarctica Systems* в 1999 году и служил в качестве представителя *Tim Berners-Lee* в *W3C Technical Architecture Group* в 2002–2004 годах. В настоящее время он занимает должность директора подразделения веб-технологий в компании *Sun Microsystems*, издает популярный веб-блог и является сопредседателем рабочей группы *IETF AtomPub Working Group*.

Брайан Кэнтрill (Bryan Cantrill) — известный разработчик компании *Sun Microsystems*, в которой большую часть своей деятельности он посвятил разработке ядра Solaris. В последнее время он со своими коллегами Майком Шапиро (Mike Shapiro) и Адамом Левенталем (Adam Leventhal) спроектировал и разработал DTrace, средство динамического инструментального оснащения производственных систем, выигравшее первую премию по инновациям «Wall Street Journal» в 2006 году.

Дуглас Крокфорд (Douglas Crockford) – выходец из американской национальной системы образования. Зарегистрированный избиратель, владелец собственного автомобиля. Разработчик систем автоматизации учрежденческой деятельности. Исследователь игровых и музыкальных возможностей системы *Atari*. Был директором технологической службы компании *Lucasfilm Ltd.* Работал директором *New Media* компании *Paramount*. Был основателем и генеральным директором компании *Electric Communities*. Был основателем и руководителем технического отдела компании *State Software*, в которой он изобрел технологию JSON. В настоящее время является разработчиком компании *Yahoo! Inc.*

Роджерио Эйтэм де Карвальо (Rogerio Atem de Carvalho) – преподаватель и исследователь в Федеральном центре технологического образования в городе Кампос (*Federal Center for Technological Education of Campos – CEFET Campos*), Бразилия. В 2006 году был отмечен наградой Международной федерации по обработке информации за выдающиеся достижения – IFIP Distinguished Academic Leadership Award в Вене, Австрия, за свои исследования по свободной (открытой) системе планирования ресурсов предприятия (Free/Open Source Enterprise Resources Planning – ERP). Его исследовательские и консультационные интересы включают также системы поддержки принятия решений – Decision Support Systems и разработку программного обеспечения.

Джефф Дин (Jeff Dean) стал работать в компании *Google* в 1999 году, и в настоящее время является научным сотрудником *Google* в группе инфраструктуры систем (*Systems Infrastructure Group*) этой компании. Работая в компании, он занимался для *Google* разработкой систем просмотра, индексирования, обслуживания запросов и размещения рекламы, разработал ряд улучшений для системы поиска и участвовал в создании ряда различных компонентов инфраструктуры распределенного вычисления для *Google*. До того как прийти в *Google*, он был сотрудником лаборатории *Western Research Laboratory* компании *DEC/Compaq*, где работал над созданием профилирующих инструментальных средств, архитектуры микропроцессоров и осуществлял поиск нужной информации. Он получил степень кандидата наук Вашингтонского университета в 1996 году, работал с Крэйгом Чамберсом (*Craig Chambers*) над технологией оптимизации компилирования для объектно-ориентированных языков. До аспирантуры он работал в глобальной программе против СПИДа во Всемирной организации здравоохранения.

Джек Донгарра (Jack Dongarra) получил степень бакалавра наук по математике в Чикагском государственном университете в 1972 году и степень магистра наук по вычислительной технике в Иллинойском институте технологии в 1973 году. Степень кандидата наук по прикладной математике он получил в Университете Нью-Мехико в 1980 году. До 1989 года он работал в *Argonne National Laboratory*, где стал старшим специалистом. Теперь он занимает должность ведущего профессора по вычислительной технике на кафедре информатики Университета Теннесси. Является ведущим сотрудником по исследованиям в подразделении по вычислительной технике и математике Национальной лаборатории Оук Ридж (*Oak Ridge National Laboratory – ORNL*), действительным членом научного общества Тьюринга в школах информатики и мате-

матики Манчестерского университета и адъюнкт-профессором кафедры информатики Университета Райса. Д. Донгарра специализируется на числовых алгоритмах линейной алгебры, параллельных вычислениях, использовании передовых компьютерных архитектур, методологии программирования и разработке инструментальных средств для параллельных компьютеров. Его исследования включают разработку, тестирование и документирование высококачественного математического программного обеспечения. Он внес свой вклад в проектирование и реализацию следующих программных пакетов и систем с открытым кодом: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS и PAPI. Он опубликовал около 200 статей, докладов, отчетов и технических заметок и являлся соавтором нескольких книг. Он был удостоен награды IEEE Sid Fernbach Award в 2004 году за свой вклад в применение высокоеффективных компьютеров с использованием инновационных подходов. Он является действительным членом научных обществ AAAS, ACM и IEEE, и членом национальной технической академии (National Academy of Engineering).

P. Кент Дибвиг (R. Kent Dybvig) является профессором информатики в Университете Индианы. Степень кандидата наук он получил в Университете Северной Каролины в 1987 году, спустя два года после того как стал сотрудником факультета в Индиане. Его исследования в области проектирования и разработки языков программирования привели к существенному вкладу в операторы управления, синтаксические абстракции, анализ программ, оптимизацию компиляторов, распределение регистров, мультипотоковое вычисление и автоматическое управление памятью. В 1984 году он создал Chez Scheme и по-прежнему остается его основным разработчиком. Известный своей быстрой компиляцией и надежностью, а также возможностью эффективно выполнять даже сложные программы, использующие большие объемы памяти, Chez Scheme был использован для создания коммерческих систем для объединения предприятий, организации веб-служб, работы с виртуальной реальностью, автоматизированной проверки лекарственных препаратов, компоновки электронных схем и многое другое. Он также используется для обучения информатике на всех уровнях и для проведения исследований в различных областях. Дибвиг является автором книги «The Scheme Programming Language», Third Edition (MIT Press) и редактором выходящего вскоре исправленного описания языка Scheme.

Майкл Фезерс (Michael Feathers) является консультантом при *Object Mentor*. Последние семь лет активно работал в сообществе гибкой разработки ПО (Agile community), разрываясь между работой с различными командами по всему миру, их обучением и тренировкой. До того как присоединиться к *Object Mentor*, Майкл разработал собственный язык программирования и написал для него компилятор.

Он также разработал мультиплатформенную библиотеку классов и рабочую среду для управления оборудованием. Совместно с другими специалистами Майкл разработал CppUnit, начальную версию переноса JUnit на C++; и FitCpp, версию платформы для проведения комплексного теста – FIT для C++. В 2005 году М. Фезерс написал книгу «Working Effectively with Legacy

Code» (Prentice Hall). Большую часть времени, свободного от работы в команде, он тратит на исследование способов изменения конструктивных основ крупных программ в течение длительного времени.

Карл Фогель (Karl Fogel) в 1995 году совместно с Джимом Блэнди (Jim Blandy) был соучредителем *Cyclic Software*, первой компании, предлагающей коммерческую поддержку CVS. В 1997 году Карл добавил к CVS поддержку анонимного доступа чтения данных из хранилища, позволив тем самым получить легкий доступ к коду разработки открытых проектов. В 1999 году он написал книгу «Open Source Development with CVS» (Coriolis). С 2000 по 2006 год работал на компанию *CollabNet, Inc.*, где руководил созданием и разработкой Subversion, системой управления версиями с открытым кодом, написанной CollabNet и командой энтузиастов систем с открытым кодом буквально с нуля. В 2005 году он написал книгу «Producing Open Source Software: How to Run a Successful Free Software Project» (O'Reilly; также доступную в сети по адресу <http://producingoss.com>). После непродолжительной работы в качестве специалиста по открытому коду в компании Google в 2006 году он ушел оттуда, чтобы стать штатным редактором «Question-Copyright.org». Фогель продолжает принимать участие в различных проектах с открытым кодом, включая Subversion и GNU Emacs.

Санджай Гхемават (Sanjay Ghemawat) – специалист компании Google, работающий в группе инфраструктуры систем (*Systems Infrastructure Group*). Он спроектировал и создал системы распределения памяти, системы индексирования текста, рабочий инструментарий, язык представления данных, систему RPC, реализовал систему выделения памяти – malloc и множество других библиотек. До того как прийти в Google, он был в числе исследователей центра *DEC Systems Research Center*, где работал над системой профилирования и оптимизирующим компилятором для Java, а также реализовал виртуальную машину Java. Он получил степень кандидата наук от MIT в 1995 году за реализацию объектно-ориентированных баз данных.

Ашиш Гулхати (Ashish Gulhati) является главным разработчиком Neomailbox, службы безопасности Интернета, и разработчиком Cryptonite, совместимой с OpenPGP защищенной системой электронной почты. Будучи свыше 15 лет разработчиком коммерческого программного обеспечения и одним из первых в Индии активистов по борьбе за цифровые права и экспертом F/OSS, он написал множество модулей открытого кода на языке Perl, которые доступны в сети CPAN. Написанные им в 1993–1994 годах статьи в журналах «PC Quest» и «DataQuest» были первыми в широкой индийской компьютерной прессе, которые вводили читателей в мир Free Software, GNU/Linux, сетевых решений и Интернета, за много лет до того как коммерческий Интернет стал доступен в Индии, и сыграли важную роль в формировании PC Quest Linux Initiative, что привело к распространению в Индии с 1995 года миллионов компакт-дисков с Linux. Благодаря своей пестрой коллекции переносных компьютеров, он быстро превращается в киборга.

Эллиот Расти Гарольд (Elliotte Rusty Harold) родом из Нового Орлеана, куда он периодически возвращается в поисках скромной миски супа из стручков бамии. Но в настоящее время Гарольд проживает в доме с видом на высотки

по соседству с Бруклином вместе со своей женой Бет, собакой по кличке Шейна и кошками Шармом (названным в честь «прелестного» квакра) и Маржори (названной в честь тещи). Он является адъюнкт-профессором информатики в Политехническом университете, где преподает Java, XML и объектно-ориентированное программирование. Его веб-сайт «Cafe au Lait» (<http://www.cafeaulait.org>) стал одним из самых популярных независимых веб-сайтов о Java в Интернете, а его дополнительный веб-сайт — «Cafe con Leche» (<http://www.cafeconleche.org>), стал одним из самых популярных XML-сайтов. В перечень написанных им книг входят «Java I/O», «Java Network Programming», «XML in a Nutshell» (обе O'Reilly), и «XML Bible» (Wiley). В настоящий момент он работает над библиотекой XOM Library для обработки XML на языке Java, движком Jaxen XPath и медиаплеером Amateur.

Брайан Хэйес (Brian Hayes) ведет колонку «Computing Science» в журнале «American Scientist», а также веб-блог <http://bit-player.org>. В прошлом он вел подобные колонки по математике и информатике для «Scientific American», «Computer Language» и «The Sciences». Его книга «Infrastructure: A Field Guide to the Industrial Landscape» (Norton) вышла в 2005 году.

Саймон Пейтон Джоунс (Simon Peyton Jones), магистр искусств, член британского компьютерного сообщества (MBCS), инженер по компьютерам, окончивший Тринити-колледж в Кембридже в 1980 году. После двух лет работы на производстве он провел семь лет в качестве преподавателя в Университетском колледже в Лондоне и девять лет в качестве профессора Университета в Глазго, перед тем как в 1998 году перебрался в Microsoft Research. Его основные исследовательские интересы распространяются на функциональные языки программирования, их реализацию и применение. Он вел ряд научно-исследовательских работ, направленных на проектирование и разработку пригодных для промышленного применения систем на функциональных языках программирования, как для однопроцессорных, так и для параллельных машин. Он внес решающий вклад в создание ставшего уже стандартом языка Haskell и стал ведущим разработчиком широко используемого компилятора Glasgow Haskell Compiler (GHC). Он написал два учебника по реализации функциональных языков.

Джим Кент (Jim Kent) — ученый-исследователь в *Genome Bioinformatics Group* в Калифорнийском университете Санта-Круз. Профессиональным программированием Кент занимается с 1983 года. Первую половину своей творческой деятельности он посвятил программному обеспечению для создания анимированных и простых изображений, став среди всего прочего автором отмеченных наградами работ Aegis Animator, Cyber Paint и Autodesk Animator. В 1996 году, устав возиться с Windows API, он решил заняться биологией, и в 2002 году получил степень кандидата наук. В качестве аспиранта он написал GigAssembler — программу, которая выдала первую сборку генома человека — за день до того, как это было сделано в компании Celera, чем помог гарантировать, что сборка генома останется свободной от патентов и других юридических заморочек. Кент является автором сорока научных работ. Его сегодняшняя работа заключается в основном в создании программ, баз данных и веб-сайтов, помогающих ученым вести анализ и разбираться со строением генома.

Брайен Керниган (Brian Kernighan) получил степень бакалавра наук в Университете Торонто в 1964 году и степень кандидата наук по электронике в Принстоне в 1969 году. Он работал в центре компьютерных исследований – *Computing Science Research* компании *Bell Labs* вплоть до 2000 года, а теперь работает на кафедре информатики в Принстоне. Он является автором восьми книг и ряда технических работ, а также держателем четырех патентов. В круг его исследовательских интересов включены языки программирования, инструментальные средства и интерфейсы, облегчающие работу с компьютерами главным образом для тех, кто не является специалистом в этой области. Он также заинтересован вопросами технического образования аудитории нетехнической направленности.

Адам Колава (Adam Kolawa) – соучредитель и генеральный директор компании *Parasoft*, ведущий поставщик программных решений автоматизированного предупреждения ошибок (*Automated Error Prevention* – АЕР). Накопленный с годами опыт по разработке различного программного обеспечения вылился в уникальную способность проникновения в суть высокотехнологичного производства и сверхъестественную способность успешно определять технологические тенденции. В результате этого он управлял разработкой нескольких успешных коммерческих продуктов, отвечающих растущим производственным потребностям повышения качества программного обеспечения, зачастую еще до того, как положенные в их основу тенденции приобретали широкое признание. Колава был соавтором книги «*Bulletproofing Web Applications*» (*Hungry Minds*), внес свой вклад в написание и собственноручно написал более 100 комментариев и технических статей для публикации в таких изданиях, как «*The Wall Street Journal*», «*CIO*», «*Computerworld*», «*Dr. Dobb's Journal*» и «*IEEE Computer*»; он также является автором ряда научных работ по физике и параллельной обработке данных. В последнее время часто выступает в таких медиа-компаниях, как CNN, CNBC, BBC и NPR. Колава имеет степень кандидата наук по теоретической физике, полученную в Калифорнийском технологическом институте, и является обладателем 10 патентов за свои последние изобретения. В 2001 году Колава был награжден ежегодной премией в категории программного обеспечения – *Los Angeles Ernst & Young's Entergeneus*.

Грег Кроа-Хартман (Greg Kroah-Hartman) является действующим специалистом по техническому обслуживанию ядра системы Linux, охватывая чуть большее количество ведущих подсистем, чем ему хотелось бы принять под свою опеку наряду с драйверным ядром и программными кодами sysfs, kobject, kref и debugfs. Он также помог становлению проектов linux-hotplug и udev и ввел в курс дела добрую половину всей команды по поддержанию стабильной работы ядра системы. Он сотрудничал с *SuSE Labs/Novell*, выполняя различные работы, связанные с ядром системы. Является автором книги «*Linux Kernel in a Nutshell*» (*O'Reilly*) и соавтором книги «*Linux Device Drivers*», Third Edition (*O'Reilly*).

Эндрю Кучлинг (Andrew Kuchling) имеет 11-летний опыт разработчика программного обеспечения и много лет принадлежит сообществу разработчиков на языке Python. К его работам, связанным с языком Python, относятся

создание и поддержка нескольких стандартных библиотечных модулей, написание серии статей «What's new in Python 2.x» и ряда другой документации, планирование проведения в 2006 и 2007 годах конференций PyCon и директорство в *Python Software Foundation*. В 1995 году Эндрю окончил Университет МакГилл со степенью бакалавра наук по информатике. Адрес его веб-сайта – <http://www.amk.ca>.

Петр Лушек (Piotr Luszczek) получил степень магистра наук в Университете горнодобывающей промышленности и металлургии в Кракове, Польша, за работы по созданию библиотек по параллельным вычислениям, не зависящим от ядра вычислительной системы. Он получил степень доктора за инновационное применение плотных матричных вычислительных ядер в разреженных направлениях и итеративных числовых алгоритмов линейной алгебры. Он применил этот опыт при разработке отказоустойчивых библиотек, использующих технологии, не зависящие от применяемых ядер вычислительных систем. В настоящее время является профессором по исследовательским работам в Университете штата Теннесси, Кноксвиль. Его работа относится к стандартизации тестирования крупных суперкомпьютерных сборок. Он является автором самонастраивающихся программных библиотек, которые автоматически выбирают наилучший алгоритм для эффективного использования доступного аппаратного обеспечения и способны вести оптимальную обработку поступающих данных. Он также занят разработкой и реализацией высокопроизводительных языков программирования.

Рональд Мак (Ronald Mak) был старшим научным сотрудником научно-исследовательского института передовой информатики (*Advanced Computer Science*), когда заключил контракт с *NASA Ames* на работу в качестве ведущего проектировщика и разработчика связующего программного обеспечения для объединенного информационного портала – *Collaborative Information Portal*. После доставки марсоходов на Марс он обеспечивал поддержку миссии в подразделениях JPL и Ames. Затем был назначен академиком Калифорнийского университета Санта-Круз и заключил новый контракт с *NASA*, на этот раз на проектирование и разработку специализированного программного обеспечения, содействующего возвращению астронавтов на Луну. Рон является сооснователем и главным техническим директором *Willard & Lowe Systems, Inc.* (www.willardlowe.com), консультационной компании, специализирующейся на информационных управляющих системах предприятия. Он написал несколько книг по компьютерному программному обеспечению и является обладателем научной степени по математике и информатике Стэнфордского университета.

Юкихиро Мацумото (Yukihiro Matsumoto, «Matz») программист, японский приверженец открытого кода и создатель набравшего в последнее время популярность языка Ruby. Он приступил к разработке Ruby в 1993 году, то есть по сути этот язык не моложе Java. Теперь он работает на компанию *Network Applied Communication Laboratory, Inc.* (*NaCl*, также известной как *netlab.jp*), спонсирующей разработку Ruby с 1997 года. Поскольку его настоящее имя является чересчур длинным для запоминания и трудным в произношении для тех, кто не владеет японским языком, в сети он пользуется псевдонимом Мац (Matz).

Арун Мехта (Arun Mehta) — инженер по электронике и компьютерный специалист, учился и преподавал в Индии, США и Германии. Он один из самых первых индийских активистов по телекоммуникации и кибернетике, добивавшийся режима наибольшего благоприятствования для пользователей, который помог бы распространить современные средства коммуникации в сельской местности и среди малообеспеченных слоев населения. Его текущие увлечения включают деревенское радио и технологии для людей с ограниченными возможностями. Он является профессором и руководителем кафедры вычислительной техники (Computer Engineering Department) в JMIT, Радор, Харьяна, Индия. Он содержит несколько веб-сайтов: <http://india-gii.org>, <http://radiophony.com> и <http://holisticit.com>.

Рафаэль Моннерат (Rafael Manhaes Monnerat) является IT-аналитиком компании *CEFET CAMPOS* и заграничным консультантом компании *Nexedi SARL*. Круг его интересов составляют свободные системы с открытым кодом, ERP и передовые языки программирования.

Трэвис Олифант (Travis E. Oliphant) получил степень бакалавра наук по электронике, компьютерным разработкам и математике в Университете Бригама Янга в 1995 году и магистра наук по электронике и компьютерным разработкам в том же учебном заведении в 1996 году. В 2001 году он получил степень кандидата наук в области прикладной биомедицины в аспирантуре Мейо в Рочестере, Миннесота. Он является ведущим автором SciPy и NumPy, научных компьютерных библиотек для языка Python. Его исследовательские интересы простираются на отображение микромасштабного импеданса, реконструкцию MRI в неоднородных полях и общие проблемы биомедицинской инверсии. В настоящее время он является доцентом кафедры электроники и вычислительной техники в Университете Бригама Янга.

Энди Орам (Andy Oram) является редактором O'Reilly Media. Сотрудничая с компанией с 1992 года, в настоящее время он специализируется на свободном программном обеспечении и технологиях с открытым кодом. Его работы в O'Reilly включают самые первые американские издания по Linux и книгу «Peer-to-Peer» в 2001 году. Скромное мастерство программиста и системного администратора он приобрел в основном самостоятельно. Энди также является членом общества «Компьютерные профессионалы за социальную ответственность» и часто пишет для O'Reilly Network (<http://oreillynet.com>) и других изданий, освещая вопросы политики, относящиеся к Интернету, и тенденции технических инноваций и их влияния на общество. Его веб-сайт расположен по адресу: <http://www.praxagora.com/andyo>.

Уильям Отте (William R. Otte) является аспирантом кафедры электротехники и информатики (*Electrical Engineering and Computer Science – EECS*) Вандербильтского университета в Теннесси. Его исследования сфокусированы на связующем программном обеспечении для распределенных и внедренных систем реального времени (*distributed real-time and embedded – DRE systems*). В настоящее время он занят некоторыми аспектами разработки механизма развертывания и конфигурации (*Deployment and Configuration Engine – DAnCE*) для компонентов CORBA. Эта работа касается исследования методик для динамического планирования и адаптации к компонентно-ориентированным приложениям

а также техническим условиям и требованиям к качественному обслуживанию приложения и отказоустойчивости. Перед поступлением в аспирантуру Уильям целый год проработал в качестве штатного инженера в Институте объединенных программных систем, куда попал по выпуску из Вандербильтского университета в 2005 году, получив степень бакалавра наук по информатике.

Эндрю Патцер (Andrew Patzer) является руководителем программы биоинформатики медицинского колледжа в Висконсине. Последние 15 лет Эндрю занимался разработкой программного обеспечения и написал ряд книг и статей, включая «Professional Java Server Programming» (Peer Information, Inc.) и «JSP Examples and Best Practices» (Apress). Его текущие интересы распространяются на вопросы биоинформатики, использование таких динамических языков, как Groovy, для извлечения громадного количества доступных биологических данных и помощи в проведении их анализа в научных целях.

Чарльз Петцольд (Charles Petzold) — свободный писатель, специализирующийся на прикладном программировании в среде Windows. Он является автором книги «Programming Windows» (Microsoft Press), которая выдержав пять изданий в период с 1988 по 1999 год, обучила использованию Windows API целое поколение программистов. Его последняя книга — «Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation» (Microsoft Press). Он также является автором уникального исследования по цифровой технологии под названием «Code: The Hidden Language of Computer Hardware and Software» (Microsoft Press). Его веб-сайт находится по адресу: <http://www.charlespetzold.com>.

Т. Раман (T. V. Raman) специализируется на веб-технологиях и озвученных пользовательских интерфейсах. В начале 90-х годов прошлого столетия он представил идею аудиоформатирования в тезисах к своей кандидатской диссертации по теме «AsTeR: Audio System For Technical Readings», где шла речь о системе, осуществлявшей высококачественное звуковое отображение технических документов. Результатом применения этих идей к более широкой сфере компьютерных пользовательских интерфейсов стала система Emacspeak. В настоящее время Раман является специалистом по исследованиям компании Google, где его усилия сконцентрированы в сфере веб-приложений.

Альберто Савойя (Alberto Savoia) является соучредителем и главным техническим директором компании *Agitar Software*. До появления в *Agitar* он был заведующим отделом разработок компании *Google*, а до этого он был заведующим отделом программных исследований в компании *Sun Microsystems Laboratories*. Его пристрастие и основное содержание работ и исследований лежит в области технологий разработки программного обеспечения, в частности, это касается инструментальных средств и технологий, помогающих программистам в тестировании и проверке работоспособности созданного ими программного кода на этапе проектирования и разработки.

Дуглас Шмидт (Douglas C. Schmidt) является профессором, доктором наук, кафедры электротехники и информатики (EECS), объединенной кафедры информатики и разработки программного обеспечения и старшим исследователем в Институте программных интегрированных систем (*Institute for Software Integrated Systems — ISIS*) Вандербильтского университета в Теннесси. Он еще раньше — специалистом по распределенным вычислительным шаблонам и структу-

рам связующего программного обеспечения, им выпущено более 350 технических трудов и написано 9 книг, охватывающих широкий диапазон тем, включая высокопроизводительные коммуникационные программные системы, параллельную обработку данных для высокоскоростных сетевых протоколов, распределенные объектные вычисления реального времени, объектно-ориентированные шаблоны для параллельных и распределенных систем и модельно-управляемые инструментальные средства разработки. В дополнение к своим академическим исследованиям доктор Шмидт является главным техническим директором компании *PrismTechnologies* и имеет более чем пятнадцатилетний опыт ведения разработки популярных связующих программных платформ с открытым кодом, содержащих богатый набор компонентов и проблемно-ориентированных языков, реализующих ключевые шаблоны для высокопроизводительных распределенных систем. Доктор Шмидт получил свою ученую степень по информатике в Калифорнийском университете в Ирвинге в 1994 году.

Кристофер Сейвальд (Christopher Seiwald) является автором *Perforce* (системы управления конфигурацией программного обеспечения), *Jam* (инструмента создания программ) и «The Seven Pillars of Pretty Code» (статьи, из которой взяты идеи для главы 32 «Код в развитии»). Перед созданием *Perforce* он руководил группой разработки сетевых решений в *Ingres Corporation*, где трудился в течение многих лет, придавая асинхронному сетевому коду презентабельный вид. В настоящее время он является главным техническим директором компании *Perforce* и по-прежнему занимается программированием.

Диомидис Спинеллис (Diomidis Spinellis) является доцентом кафедры управления научными и техническими исследованиями Афинского университета экономики и бизнеса, Греция. Его научные интересы включают инструментальные средства разработки программного обеспечения, языки программирования и безопасность компьютерных систем. Он имеет степени магистра технических наук в области информатики и кандидата наук в этой же области, обе степени получены в Имперском колледже в Лондоне. Он издал более ста технических работ в области разработки программного обеспечения, информационной безопасности и так называемых повсеместных вычислений (*ubiquitous computing*). Спинеллис также написал две книги из серии «Open Source Perspective»: «Code Reading» (за которую удостоен премии Software Development Productivity Award 2004) и «Code Quality» (обе книги выпущены издательством Addison-Wesley). Он является членом редколлегии «IEEE Software», ведущим регулярной колонки «Tools of the Trade». Спинеллис – приверженец FreeBSD и автор ряда пакетов, библиотек и инструментальных средств с открытым кодом.

Линкольн Стейн (Lincoln Stein) является доктором медицины и философии, работающим над компоновкой и визуализацией биологических данных. После обучения в Гарвардском медицинском институте он работал в центре генетических исследований *Whitehead Institute/MIT Center for Genome Research*, где разработал базы данных, используемые для составления карт геномов мыши и человека. В лаборатории *Cold Spring Harbor* он работал над различными базами данных градаций генома, включая WormBase, базу данных генома свободноживущейся нематоды; Gramene, сравнительную базу данных отображения генома для риса и других однодольных растений; базу данных международного проекта International Hap-Map Project Database и базу данных биологических процес-

сов человека, названную Reactome. Линкольн является также автором книг «How to Set Up and Maintain a Web Site» (Addison-Wesley), «Network Programming in Perl» (Addison-Wesley), «Official Guide to Programming with CGI.pm» (Wiley) и «Writing Apache Modules with Perl and C» (O'Reilly).

Невин Томпсон (Nevin Thompson) перевел с японского главу 29 «Отношение к коду как к очерку», написанную Юкихиро Мацумото (Yukihiro Matsumoto). Среди его клиентов крупнейшая японская телевизионная сеть, а также *Technorati Japan* и *Creative Commons*.

Генри Уоррен-мл. (Henry S. Warren, Jr.) в течение 45 лет работал в компании *IBM*, в период развития компьютерной техники от *IBM 704* до *PowerPC*. Он создавал различные военные системы контроля и управления, работал в Нью-Йоркском университете над проектом *SETL* под руководством Джека Шварца (Jack Schwartz). С 1973 года работал в отделе исследований *IBM*, занимаясь в основном компиляторами и компьютерными архитектурами. В настоящее время он работает над компьютерным проектом *Blue Gene petaflop*. Степень кандидата наук по информатике получил в Курантовском институте Нью-Йоркского университета. Он является автором книги «*Hacker's Delight*» (Addison-Wesley).

Лаура Уингерд (Laura Wingerd) сформировала свой взгляд на управление конфигурациями программного обеспечения (SCM) в течение десятилетнего ожесточенного отстаивания встраиваемого и исходного кода для таких продуктов, как СУБД *Sybase* и *Ingres*. Она стала работать в *Perforce Software* в первый же год существования компании и с тех пор приобрела достаточный опыт управления конфигурациями программного обеспечения благодаря тем самым клиентам *Perforce*, которых она взялась консультировать. Она является автором книги «*Practical Perforce*» (O'Reilly) и ряда связанных с SCM докладов. Ее видеодебют был отмечен технической лекцией *Google* «The Flow of Change». В настоящее время Лаура является вице-президентом, отвечающим за программную технологию компании *Perforce Software*, распределяя свое время между внедрением обычной практики использования SCM и исследованием новых, более эффективных способов использования *Perforce*.

Грег Уилсон (Greg Wilson) получил степень кандидата наук в области информатики в Эдинбургском университете и работал над высокопроизводительными научными вычислениями, визуализацией данных и компьютерной безопасностью. Теперь он работает адъюнкт-профессором кафедры информатики Университета Торонто и пишущим редактором журнала «*Dr. Dobb's Journal*».

Андреас Целлер (Andreas Zeller) — выпускник Дармштадтского технического университета 1991 года, в 1997 году получил степень кандидата наук в области информатики в Техническом университете Брауншвейга в Германии. С 2001 года — профессор информатики в Университете Саарланда, Германия. Целлер исследует большие программы и их историю и является разработчиком ряда методов по определению причин отказов, возникающих в открыто распространяемых программах, а также в индустриальном окружении *IBM*, *Microsoft*, *SAP* и других. Его книга «*Why Programs Fail: A Guide to Systematic Debugging*» (Morgan Kaufmann) получила в 2006 году премию *Software Development Magazine productivity award*.