

Оглавление

1.	Система Asterisk.....	14
1.1.	Основные архитектурные концепции	14
1.1.1.	Каналы.....	14
1.1.2.	Соединение каналов типа «мост».....	15
1.1.3.	Фреймы	16
1.2.	Компонентные абстракции системы Asterisk.....	16
1.2.1.	Драйверы каналов	17
1.2.2.	Приложения планирования прохождения вызовов (dialplan).....	18
1.2.3.	Функции dialplan	20
1.2.4.	Трансляторы кодеков.....	20
1.3.	Потоки	21
1.3.1.	Потоки мониторинга сети	22
1.3.2.	Канальные потоки	22
1.4.	Сценарии вызовов	22
1.4.1.	Проверка голосовой почты	22
1.4.2.	Вызов с созданием «моста»	24
1.5.	Заключительные комментарии	26
	Примечания	26
2.	Audacity	26
2.1.	Структура Audacity	27
2.2.	Библиотека графического интерфейса wxWidgets	30
2.3.	Слой ShuttleGui	31
2.4.	Панель TrackPanel	32
2.5.	Библиотека PortAudio: запись и воспроизведение.....	34
2.6.	Блочные файлы BlockFile.....	36
2.7.	Использование сценариев	38
2.8.	Эффекты режима реального времени	40
2.9.	Заключение	40
	Примечания	41
3.	Командная оболочка Bourne-Again Shell.....	41
3.1.	Ведение	41
3.1.1.	Bash.....	42
3.2.	Синтаксические единицы и примитивы	43
3.2.1.	Примитивы	43
3.2.2.	Переменные и параметры.....	43
3.2.3.	Язык программирования командной оболочки	44
3.2.4.	Дополнительное замечание.....	45
3.3.	Обработка входных данных	45

3.3.1. Readline и редактирование командной строки	45
3.3.2. Обработка входных данных в неинтерактивном режиме	47
3.3.3. Многобайтовые символы	48
3.4. Анализ	48
3.5. Раскрытие слов	50
3.5.1. Раскрытие параметров и переменных	50
3.5.2. И многое другое	50
3.5.3. Разбиение на слова	51
3.5.4. Подстановка	51
3.5.5. Реализация	52
3.6. Исполнение команд	52
3.6.1. Перенаправление	52
3.6.2. Встроенные команды	53
3.6.3. Выполнение простых команд	54
3.6.4. Управление заданиями	54
3.6.5. Составные команды	55
3.7. Усвоенные уроки	56
3.7.1. Что, по моему мнению, важно	56
3.7.2. Что я должен был бы сделать по-другому	56
3.8. Заключение	57
Примечания:	57
4. Архитектура приложений с открытым исходным кодом. Том 1. Глава 4. Berkeley DB	57
4.1. В начале	58
4.2. Обзор архитектуры	59
4.3. Методы доступа: Btree, Hash, Recno, Queue	64
4.4. Интерфейсный слой библиотеки	65
4.5. Компоненты, лежащие глубже	66
4.6. Менеджер буферирования: Mpool	68
4.6.1. Файловая абстракция Mpool	68
4.6.2. Запись в журнал с упреждением	68
4.7. Менеджер блокировок: Lock	69
4.7.1. Блокируемые объекты	70
4.7.2. Матрица конфликтов	70
4.7.3. Поддержка иерархической блокировки	71
4.8. Менеджер журнала: Log	72
4.8.1. Формат журнальных записей	73
4.8.2. Разрушая абстракцию	74
4.9. Менеджер транзакций: Txn	75
4.9.1. Обработка контрольных точек	76
4.9.2. Восстановление	77
4.10. Заключение	78

Примечания	78
5. CMake.....	79
5.1. История создания CMake и предъявленные к нему требования	79
5.2. Как реализован CMake	82
5.2.1. Процедура использования CMake	82
5.2.2. CMake: Код	83
5.2.3. Графические интерфейсы.....	86
5.2.4. Тестирование CMake	87
5.3. Усвоенные уроки.....	87
5.3.1. Обратная совместимость	87
5.3.2. Язык, язык, язык.....	87
5.3.3. Плагины не работают	88
5.3.4. Ограничение распространения API.....	88
6. Среда разработки Eclipse.....	88
6.1. Ранний вариант Eclipse	89
6.1.1. Платформа	90
6.1.2. Инструментарий Java Development Tools (JDT).....	97
6.1.3. Среда разработки плагинов PDE	97
6.2. Eclipse 3.0: Среда времени выполнения, RCP и роботы	98
6.2.1. Среда времени выполнения	98
6.2.2. Платформа Rich Client Platform (RCP).....	101
6.3. Eclipse 3.4	103
6.3.1. Концепции p2	105
6.4. Eclipse 4.0	107
6.4.1. Рабочее пространство модели.....	108
6.4.2. Стиль каскадных стилевых страниц	108
6.4.3. Внедрение зависимостей	109
6.4.4. Сервисы приложений	111
6.5. Заключение	111
Примечания	111
7. Проект Graphite	112
7.1. Библиотека базы данных: хранение данных временных рядов.....	112
7.2. Серверная часть: Простой сервис хранения данных	113
7.3. Клиентская часть: Графики по запросу	114
7.4. Панели управления	115
7.5. Очевидное узкое место	115
7.6. Оптимизация ввода/вывода.....	116
7.7. Все это в режиме реального времени.....	117
7.8. Ядра, кэширование и катастрофические отказы.....	118
7.9. Кластеризация	119
7.9.1. Краткий анализ эффективности кластеризации.....	119

7.9.2. Хранение метрик в кластере	120
7.10. Размышления о проекте	120
7.11. Переход в статус открытого кода.....	121
Примечания	122
8. Распределенная файловая система Hadoop	122
8.1. Введение	122
8.2. Архитектура.....	123
8.2.1. Сервер метаданных NameNode.....	123
8.2.2. Образ и журнал.....	123
8.2.3. Сервер данных приложений DataNode	124
8.2.4. Клиент HDFS	125
8.2.5. Сервер файлов контрольных точек CheckpointNode	126
8.2.6. Сервер резервных копий BackupNode	126
8.2.7. Обновления и снимки файловой системы	127
8.3. Операции ввода/вывода и управление копиями блоков	128
8.3.1. Чтение и запись файлов.....	128
8.3.2. Размещение блоков	130
8.3.3. Управление репликацией	131
8.3.4. Балансировщик.....	132
8.3.5. Сканер блоков	133
8.3.6. Прекращение эксплуатации серверов данных приложений.....	133
8.3.7. Копирование данных между кластерами.....	133
8.4. Практическое использование файловой системы в компании Yahoo!	133
8.4.1. Долговечность хранения данных.....	134
8.4.2. Возможности совместного использования ресурсов HDFS	135
8.4.3. Масштабирование и объединение файловой системы	135
8.5. Выученные уроки.....	136
8.6. Благодарности	137
Сноски	137
9. Непрерывная интеграция	137
9.1. Многообразие систем непрерывной интеграции	138
9.1.1. Какие функции выполняются программным обеспечением непрерывной интеграции	138
9.1.2. Внешние взаимодействия.....	140
9.2. Архитектуры.....	141
9.2.1. Модель реализации: Buildbot	141
9.2.2. Модель реализации: CDash	142
9.2.3. Модель реализации: Jenkins	143
9.2.4. Модель реализации: Pony-Build	144
9.2.5. Рецепты сборки	146
9.2.6. Управление доверенными ресурсами	146
9.2.7. Выбор модели	146

9.3. Будущее систем непрерывной интеграции.....	147
9.3.1. Заключительные размышления	147
9.3.2. Благодарности	148
10. Фреймворк Jitsi.....	149
10.1. Разработка Jitsi	149
10.2. Jitsi и фреймворк OSGi	150
10.3. Собираем и запускаем сборку	153
10.4. Сервис провайдера протоколов	154
10.4.1. Наборы операций	155
10.4.2. Аккаунты, фабрики и экземпляры провайдеров	156
10.5. Медиасервис	156
10.5.1. Захват медиа, потоковая обработка и воспроизведение	157
10.5.2. Кодеки	158
10.5.3. Подключение к провайдерам протоколов	158
10.6. Сервис пользовательского интерфейса.....	159
10.7. Усвоенные уроки.....	161
10.7.1. Звук Java Sound и звук PortAudio	161
10.7.2. Захват и рендеринг видео.....	162
10.7.3. Кодирование и декодирование видео	162
10.7.4. Прочее	163
10.8. Благодарности	163
Примечания	163
Сноски	163
11. LLVM	164
11.1 Краткое описание классической архитектуры компиляторов.....	164
11.1.1 Последствия использования данной архитектуры	165
11.2. Существующие реализации языков программирования.....	166
Сноски	167
11.3. Представление кода в LLVM: LLVM IR	167
11.3.1. Разработка алгоритмов оптимизаций для LLVM IR	169
Сноски	170
11.4. Реализация архитектуры трех фаз в LLVM.....	170
11.4.1 Представление LLVM IR является завершенным представлением кода	171
11.4.2. LLVM является набором библиотек	171
Сноски	173
11.5. Архитектура многоцелевого генератора кода LLVM	174
11.5.1. Файлы описания целевой архитектуры в LLVM	174
11.6. Интересные возможности, предоставляемые модульной архитектурой.....	176
11.6.1. Выбор момента и порядка выполнения каждой из фаз	176
11.6.2. Тестирование элементов оптимизатора	177
11.6.3. Автоматическое тестирование с помощью BugPoint	178

Сноски	179
11.7. Взгляд в прошлое и направления развития в будущем	179
Сноски	180
12. Архитектура Mercurial	180
12.1. Краткая история контроля версий	180
12.1.1. Централизованный контроль версий	181
12.1.2. Распределенный контроль версий	182
12.2. Структуры данных	183
12.2.1. Проблемы	184
12.2.2. Быстрое хранение версий: Revlogs	184
12.2.3. Три revlog	185
12.2.4. Рабочая копия	187
12.3. Механизм контроля версий	187
12.3.1. Ветки	187
12.3.2. Тэги	188
12.4. Общая структура	189
12.5. Расширяемость	191
12.5.1. Написание расширений	191
12.5.2. Перехватчики	192
12.6. Выводы	192
13. Экосистема NoSQL	193
13.1. Что в имени?	194
13.1.1. Язык SQL и реляционная модель	194
13.1.2. Исходные данные для проектирования систем NoSQL	195
13.1.3. Характеристики и соображения	196
13.2. Модели данных и запросов систем NoSQL	197
13.2.1. Модели данных систем NoSQL на основе ключей	197
13.2.2. Хранилища на основе графов	199
13.2.3. Сложные запросы	199
13.2.4. Транзакции	199
13.2.5. Хранилище данных без жестко заданной схемы	200
13.3. Долговечность хранения данных	200
13.3.1. Долговечность хранения данных на отдельном сервере	201
13.3.2. Долговечность хранения данных на множестве серверов	202
13.4. Масштабирование с целью повышения производительности	203
13.4.1. Не фрагментируйте систему без необходимости	204
13.4.2. Фрагментация системы с помощью программ для координации запросов	205
13.4.3. Последовательные хэш-кольца	205
13.4.4. Распределение с использованием диапазонов	207
13.4.5. Какую систему распределения данных следует использовать	209
13.5. Согласованность данных	210

13.5.1. Немного о CAP	210
13.5.2. Строгая согласованность данных	211
13.5.3. Конечная согласованность данных	212
13.6. Заключительное слово	214
13.7. Благодарности	214
14. Архитектура системы управления пакетами в Python	215
14.1 Введение	215
14.2. Трудности разработчиков Python	216
14.3 Современная архитектура системы управления пакетами	218
14.3.1. Краткое описание и дефекты архитектуры Distutils.....	218
14.3 Современная архитектура системы управления пакетами	220
14.3.2. Метаданные и PyPI	220
14.3 Современная архитектура системы управления пакетами	221
14.3.3. Архитектура PyPI.....	221
14.3 Современная архитектура системы управления пакетами	224
14.3.4. Архитектура системы установки Python	224
14.3 Современная архитектура системы управления пакетами	225
14.3.5. Setuptools, Pip и аналогичные проекты	225
14.3 Современная архитектура системы управления пакетами	226
14.3.6. Как насчет файлов данных?	226
14.4 Усовершенствованные стандарты	227
14.4.1. Метаданные	227
14.4 Усовершенствованные стандарты.....	229
14.4.2. Что установлено?	229
14.4 Усовершенствованные стандарты.....	230
14.4.3. Архитектурные решения в отношении работы с файлами данных	230
14.4 Усовершенствованные стандарты	232
14.4.4. Усовершенствования каталога PyPI.....	232
14.5 Подробности реализации	235
14.6 Выученные уроки.....	236
14.6.1. О стандартах PEP	236
14.6.2. Пакет, добавленный в стандартную библиотеку, находится одной ногой в могиле ...	236
14.6.3. Обратная совместимость	237
14.7. Справочные материалы и вклад сообщества	237
Сноски	238
15. Riak и Erlang/OTP	238
15.1. Краткое введение в язык Erlang.....	238
15.2. Остов процесса	241
15.3. Поведения OTP	241
15.3.1. Введение	242
15.3.2 Основные серверы	243

15.3.3. Запуск вашего сервера	243
15.3.4. Передача сообщений.....	245
15.3.5. Остановка сервера.....	247
15.4. Другие поведения рабочих процессов	247
15.4.1. Автоматы конечных состояний	247
15.4.2. Обработчики событий	248
15.5. Супервизоры.....	249
15.5.1. Функции обратного вызова супервизора.....	249
15.5.2. Приложения	252
15.6. Репликация и коммуникация в Riak	252
15.7. Заключение и усвоенные уроки.....	253
15.7.1. Благодарности	254
16. Проект Selenium WebDriver	254
16.1. История	254
16.2. Пару слов о жаргоне	256
16.3. Вопросы архитектуры.....	256
16.3.1. Снижение расходов.....	257
16.3.2. Эмуляция поведения пользователя	257
16.3.3. Перекладывание всей работы на драйвера	257
16.3.4. Вам не нужно понимать, как это все работает	258
16.3.5. Уменьшение влияния фактора автобуса.....	258
16.3.6. Следует дружественно относиться к реализации языка Javascript	258
16.3.7. Каждый вызов — это дистанционный вызов RPC	258
16.3.8. Заключительный штрих: Это проект с открытым исходным кодом	259
16.4. Побеждаем сложность	259
16.4.1. Схема WebDriver	260
16.4.2. Комбинаторный взрыв.....	262
16.4.3. Недостатки схемы, используемой в WebDriver	263
16.5. Уровни и Javascript	263
16.6. Дистанционный драйвер и, в частности, драйвер Firefox	266
16.7. Драйвер IE.....	272
16.8. Selenium RC	276
16.9. Оглядываясь назад	278
16.10. Заглядывая в будущее.....	278
17. Sendmail - Архитектура и принципы разработки	279
17.1. Как все начиналось...	280
17.2. Принципы разработки	282
17.2.1 Один программист ограничен в своих возможностях.....	282
17.2.2. Не переделывай пользовательские агенты	282
17.2.3. Не переделывай хранилище локальной почты.....	282
17.2.4. Пусть sendmail подстраивается под мир, а не наоборот	283

17.2.5. Менять как можно меньше.....	283
17.2.6. Сразу думай о надежности	283
17.2.7. Что не было реализовано.....	283
17.3. Фазы разработки	283
17.3.1. Волна первая: delivermail	283
17.3.2. Волна 2: sendmail 3, 4 и 5	284
17.3.3. Волна 3: Годы хаоса.....	285
17.3.4 Волна 4: sendmail 8	286
17.3.5 Волна 5: Коммерческие годы.....	286
17.3.6. Что произошло с sendmail 6 и 7?	287
17.4. Проектные решения.....	287
17.4.1. Синтаксис конфигурационного файла.....	287
17.4.2. Правила преобразования (Rewriting Rules)	287
17.4.3. Использование преобразования для парсинга	288
17.4.4. Внедрение SMTP и организации очереди в sendmail	289
17.4.5. Реализация очереди.....	289
17.4.6. Получение и исправление неверного ввода	290
17.4.7. Конфигурирование и использование M4.....	291
17.5. Другие соображения	292
17.5.1. Об оптимизации масштабных Интернет-систем	292
17.5.2. Milter.....	292
17.5.3. Расписание релизов.....	293
17.6. Безопасность	293
17.7. Эволюция sendmail.....	295
17.7.1. Конфигурирование становится более подробным.....	295
17.7.2. Большее количество соединений с другими подсистемами: большая интеграция	295
17.7.3. Адаптация к враждебному миру.....	296
17.7.4. Внедрение новых технологий	296
17.8. Что если бы я делал sendmail сегодня?	297
17.8.1. Что бы я сделал иначе.....	297
17.8.2. Что я бы оставил точно так же	298
17.9. Заключение	299
18. SnowFlock	300
18.1. Знакомство с SnowFlock	300
18.2. Технология клонирования виртуальных машин	301
18.3. Подход SnowFlock	302
18.4. Архитектурный дескриптор виртуальной машины	305
18.5. Компоненты на стороне родительской виртуальной машины	306
18.5.1. Процесс сервера памяти	306
18.5.2. Многоадресное распространение данных с помощью службы mcdist	306
18.5.3. Виртуальный диск.....	307

18.6. Компоненты на стороне клонированных виртуальных машин.....	308
18.6.1. Процесс memtap	308
18.6.2. Оптимально работающие клонированные виртуальные машины избегают чрезмерных запросов	309
18.7. Интерфейс приложений для клонирования виртуальных машин.....	309
18.7.1. Реализация API.....	309
18.7.2. Необходимые изменения.....	310
18.8. Заключение	311
Сноски	312
19. Электронные таблицы SocialCalc	312
19.1. WikiCalc	313
19.2. SocialCalc	316
19.3. Цикл работы команд	318
19.4. Редактор таблиц	320
19.5. Формат сохранения save.....	321
19.6. Расширенные возможности редактирования	323
19.6.1. Типы и форматы.....	323
19.6.2. Отображение wiki-текста	324
19.7. Совместные работы в режиме реального времени	326
19.7.1. Кросс-браузерный транспорт.....	327
19.7.2. Разрешение конфликтов.....	328
19.7.3. Дистанционное управление курсором	330
19.8. Усвоенные уроки.....	331
19.8.1. Главный конструктор с ясным видением	331
19.8.2. Wiki для сообщества, работающего с проектом	332
19.8.3. Объединяем разные часовые пояса	332
19.8.4. Оптимизация для удовольствия.....	332
19.8.5. Управление разработкой с использованием «описаний - проверок»	333
19.8.6. Открытый исходный код с лицензией CPAL	333
Примечания	334
20. Фреймворк Telepathy	334
20.1. Компоненты фреймворка Telepathy	335
20.2. Как в Telepathy используется шина D-Bus	337
20.2.1. Хэндлы (Handles)	340
20.2.2. Обнаружение сервисов Telepathy	340
20.2.3. Снижение трафика шины D-Bus.....	341
20.3. Соединения, каналы и клиентские приложения	342
20.3.1. Соединения	343
20.3.2. Каналы.....	343
20.3.3. Запрос каналов, свойств каналов и диспетчеризация.....	344
20.3.4. Клиентские программы	346

20.4. Роль привязок к языкам программирования	348
20.4.1. Асинхронное программирование	348
20.4.2. Готовность объектов.....	350
20.5. Устойчивость к ошибкам	350
20.6. Расширение Telepathy: механизм sidecars	351
20.7. Беглый взгляд внутрь менеджера соединений.....	352
20.8. Усвоенные уроки.....	353
Примечания	353
21. Фреймворк Thousand Parsec	354
21.1. Анатомия звездной империи Star Empire	355
21.1.1. Объекты	356
21.1.2. Распоряжения	357
21.1.3. Ресурсы	358
21.1.4. Модели	358
21.2. Протокол Thousand Parsec	359
21.2.1. Основные фреймы.....	360
21.2.2. Игроны и игры	360
21.2.3. Объекты, распоряжения и ресурсы	362
21.2.4. Работа с Моделями	363
21.2.5. Администрирование сервера	365
21.3. Поддерживаемые функциональные возможности.....	365
21.3.1. Хранение данных на сервере	365
21.3.2. Язык компонентов Thousand Parsec Component Language.....	366
21.3.3. Описание сражения - BattleXML.....	367
21.3.4. Метасервер.....	367
21.3.5. Однопользовательский режим.....	368
21.4. Усвоенные уроки.....	370
21.4.1. Что работает	370
21.4.2. Что не работает.....	370
21.4.3. Заключение	371
Примечания	371
22. Фреймворк Violet	371
22.1. Введение в Violet.....	372
22.2. Графический фреймворк	373
22.3. Использование свойств JavaBeans.....	376
22.4. Хранение полученных результатов.....	377
22.5. Java WebStart	379
22.6. Java 2D.....	380
22.7. Это не фреймворк приложений Swing	381
22.8. Операции Undo/Redo	382
22.9. Архитектура плагинов	384

22.10. Заключение	384
Примечания	385
23. Система VisTrails	385
23.1. Обзор системы.....	387
23.1.1. Рабочие процессы и системы на базе рабочих процессов	387
23.1.3. Информация о происхождении данных и рабочих процессов	388
23.1.3. Пользовательский интерфейс и базовые функциональные возможности	388
23.2. История проекта.....	390
23.3. Внутри системы VisTrails.....	390
23.3.1. Дерево версий: возможность выбора источника данных.....	391
23.3.2. Выполнение рабочего процесса, кэширование данных	392
23.3. Внутри системы VisTrails.....	393
23.3.3. Сериализация и хранение данных	393
23.3.4. Расширяемость с помощью пакетов и языка Python	394
23.3.5. Пакеты и сборки системы VisTrails	395
23.3.6. Передача данных в виде модулей.....	397
23.4. Компоненты и возможности	398
23.4.1. Таблица визуализации	398
23.4.2. Визуальные различия и аналогии.....	399
23.4. Компоненты и возможности	401
23.4.3. Запрос информации о происхождении	401
23.4.4. Хранение данных	401
23.4.5. Обновления.....	402
23.4.6. Совместный доступ и публикация результатов, использующих информации о происхождении.....	403
23.5. Усвоенные уроки.....	404
23.5.1. Благодарности	405
Примечания	405
24. Система VTK	405
24.1. Что такое VTK?	406
24.2. Архитектурные особенности	407
24.2.1. Основные особенности.....	407
24.2.2. Представление данных	410
24.2.3. Конвейерная архитектура.....	411
24.2.4. Подсистема рендеринга.....	414
24.2.5. События и взаимодействие	416
24.2.6. Краткое описание библиотек	417
24.3. Оглядываясь назад / заглядывая вперед	418
24.3.1. Управление ростом системы.....	418
24.3.2. Технологические дополнения	419
24.3.3. Наука с открытыми возможностями	419

24.3.4. Усвоенные уроки.....	420
Примечания	420
25. Битва за Веснот	420
25.1. Обзор проекта.....	421
25.2. Wesnoth Markup Language — язык разметки Wesnoth	423
25.3. Юниты в Wesnoth.....	425
25.4. Реализация в Wesnoth многопользовательских игр.....	427
25.5. Заключение	428

1. Система Asterisk

Глава 1 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Asterisk [1] является платформой с открытым исходным кодом, распространяемой по лицензии GPLv2, которая предназначена разработки приложений телефонии. Если кратко, то это серверное приложение, с помощью которого можно делать вызовы, можно принимать вызовы и можно осуществлять специальную обработку телефонных вызовов.

Проект был запущен Марком Спенсером (Mark Spencer) в 1999 году. У Марка была компания Linux Support Services (оказывающая услуги по поддержке Линукс) и ему нужна была телефонная система, которая бы помогала вести его бизнес. У него не было достаточно денег на покупку готовой системы, поэтому он просто сделал свою собственную. По мере того, как росла популярность системы Asterisk, интересы компании Linux Support Services сместились в сторону проекта Asterisk и компания Linux Support Services была переименована в компанию Digium, Inc.

Название системы Asterisk пошло от названия символа «*» («звездочка», на английском языке - «asterisk»), который в системе Unix является универсальным символом. Целью проекта Asterisk было предоставить возможность делать все, что необходимо в телефонии. Продвигаясь к этой цели, система Asterisk теперь поддерживает длинный список технологий, применяемых для осуществления и приема телефонных вызовов. К ним относятся многие протоколы VoIP (Voice over IP - голос поверх IP), а также как аналоговые, так и цифровые подключения к традиционным телефонным сетям общего пользования PSTN (Public Switched Telephone Network). Одним из главных преимуществ системы Asterisk является ее способность осуществлять в системе или получать из системы вызовы различных типов.

Поскольку из системы Asterisk можно делать телефонные вызовы и их можно в системе принимать, есть также большое количество дополнительных возможностей, которые можно выбрать для обработки телефонных вызовов. Некоторые возможности, такие как голосовая почта, реализованы с помощью больших предварительно встроенных приложений общего назначения. Есть другие меньшие возможности, например, воспроизведение звуковых файлов, чтение клавиш с цифрами или распознавание речи, которые можно объединять друг с другом для создания собственных приложений голосовой обработки.

1.1. Основные архитектурные концепции

В этом разделе обсуждаются некоторые архитектурные концепции, являющиеся важными для всех частей системы Asterisk. Эти концепции являются фундаментом архитектуры Asterisk.

1.1.1. Каналы

Канал (channel) в Asterisk представляет собой соединение между системой Asterisk и некоторым конечным телефонным устройством (рис.1.1). Наиболее общим примером является случай, когда телефон отправляет телефонный вызов в систему Asterisk. Такое соединение будет представлено в виде в отдельного канала. В коде Asterisk канал существует как экземпляр структуры данных `{ast_channel}`.

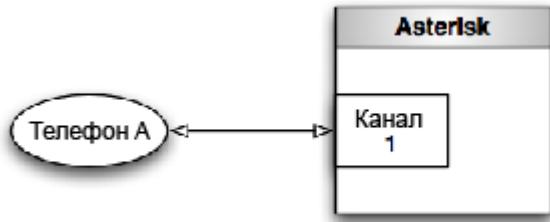


Рис.1.1: Одиночный вызов, представленный отдельным каналом

1.1.2. Соединение каналов типа «мост»

Наверное, гораздо более знаком сценарий вызова, когда соединение осуществляется между двумя телефонами: человек, использующий телефон А, звонит человеку, имеющему телефон В. В этом сценарии вызовов есть два конечных телефонных устройства, подключенных к системе Asterisk, поэтому для этого вызова существуют два канала (рис.1.2).

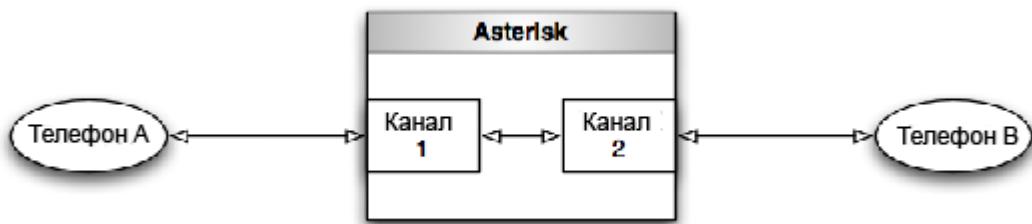


Рис.1.2: Два звена вызова, представленных двумя каналами

Когда каналы системы Asterisk соединяются таким образом, то такое соединение между каналами называется соединением типа «мост» (channel bridge). Создание моста является действием по соединению каналов вместе с целью передачи между ними медиаданных. Даже в случае, когда для каждой вызова с конечного устройства есть более одного потока медиаданных (например, аудио и видео данных), в системе Asterisk используется только единственный канал. На рис.1.2, где показаны два канала для телефонов А и В, на соединение типа «мост» возложена обязанность передачи медиаданных, идущих из телефона А в телефон В и, аналогичным образом, передачи медиаданных, идущих из телефона В в телефон А. Система Asterisk выступает в роли посредника по передаче всех потоков медиаданных. Не разрешено ничего, что не определено в системе Asterisk и над чем в системе Asterisk нет полного контроля. Это означает, что система Asterisk может делать записи, манипулировать с аудио-сообщениями и производить преобразования между форматами, относящимися к различным технологиям.

Когда происходит соединение двух каналов с помощью моста, то есть два способа, с помощью которых это можно сделать: создать универсальный мост (generic bridge) или нативный мост (native bridge). Универсальный мост является соединением, которое может работать независимо от того, какие технологии используются внутри канала. Он передает все аудио-сообщения и сигналы через абстрактный интерфейс, имеющийся в системе Asterisk. Хотя это наиболее гибкий способ создания мостового соединения, он является также наименее эффективным из-за того, что есть несколько уровней абстракций, которые требуется использовать с тем, чтобы выполнить задачу. Пример универсального моста приведён на рис.1.2.

Решение о выборе универсального или нативного мостового соединения делается с помощью сравнения каналов в тот момент, когда они соединяются мостом. Если для обоих каналов указывается, что в них поддерживается один и тот же способ нативного мостового соединения, то именно он и будет использоваться. В противном случае будет использован универсальный способ соединения.

нения типа «мост». Для того чтобы определить, поддерживается ли в обоих каналах один и тот же нативный способ мостового соединения, применяется простая функция, написанная на языке C, в которой выполняется сравнение указателей. Естественно, что это не самый элегантный способ, но мы еще ни разу не сталкивались с какими-либо случаями, когда этого для наших целей оказывалось недостаточным. Функция создания нативных мостовых соединений каналов будет более подробно рассматриваться в разделе 1.2. На рис.1.3 показан пример нативного мостового соединения.

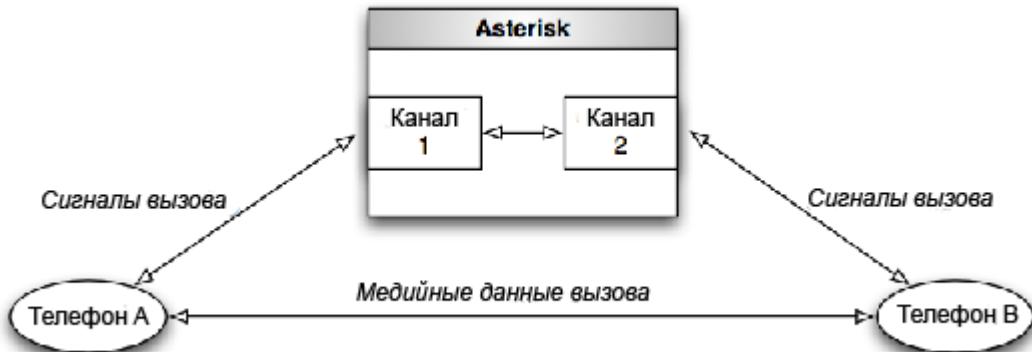


Рис.1.3. Пример нативного соединения типа «мост»

1.1.3. Фреймы

Передача данных внутри кода Asterisk во время осуществления вызова происходит с использованием фреймов (frames); каждый фрейм является экземпляром структуры данных `ast_frame`. Фреймы могут быть либо медиафреймами (media frames), либо сигнальными фреймами (signalling frames). Во время обычного телефонного вызова звонка через систему будет передаваться поток медиафреймов, содержащих аудио данные. Сигнальные фреймы используются для передачи сообщений о событиях, касающихся вызова, например, о нажатии кнопки с цифрой, о переводе вызова в режим ожидания или о завершении вызова.

Список имеющихся фреймов определен статически. Каждый фрейм маркируется числом, в котором закодирован тип и подтип. Полный список можно найти в исходном коде в файле `include/asterisk/frame.h`; ниже приведено несколько примеров:

- **VOICE**: С помощью этих фреймов осуществляется передача порции аудиоданных.
- **VIDEO**: С помощью этих фреймов осуществляется передача порции видеоданных.
- **MODEM**: Этот кодирование используется для данных при передаче данных во фрейме, например, использование протокола T.38 для передачи факса поверх IP. Первоначально этот тип фрейма использовался для факсов. Важно, что фреймы с данными можно совершенно спокойно оставлять в том виде, как они есть, поскольку сигнал будет успешно декодирован на другом конце линии. Они отличаются от фреймов AUDIO, так что его можно преобразовывать с помощью других кодеков и сохранять пропускную способность канала за счет снижения качества звука.
- **CONTROL**: Сообщение, сигнализирующее о вызове, которое указывается этим фреймом. Эти фреймы используются для индикации событий, сигнализирующих о вызове. К числу этих событий относятся ответ на телефонный вызов, завершение вызова, удержание вызова и т.д.
- **DTMF_BEGIN**: Начало нажатия на клавишу с цифрой. Этот фрейм отправляется, когда абонент начинает нажимать клавишу DTMF [2] на своем телефоне
- **DTMF_END**: Завершение нажатия на клавишу с цифрой. Этот фрейм отправляется, когда абонент прекращает нажимать клавишу DTMF [2] на своем телефоне.

1.2. Компонентные абстракции системы Asterisk

Asterisk является приложением с высокой степенью модульности. Есть базовая часть приложения, которая собирается в каталоге исходного кода `main/`. Но она, сама по себе, будет мало чем для вас полезной. Базовая часть приложения выступает, прежде всего, в роли реестра модулей. В базовой части также есть код, в котором указано, как соединять все абстрактные интерфейсы вместе для того, чтобы можно было пропускать через систему телефонные вызовы. Конкретные реализации этих интерфейсов будут регистрироваться модулями, загружаемыми на этапе выполнения программы.

Когда запускается базовая часть приложения, то по умолчанию загружаются все модули, найденные в предопределенных каталогах файловой системы. Этот подход был выбран как самый простой. Тем не менее, есть отдельный конфигурационный файл, в котором можно уточнить, какие модули и в каком порядке следует загружать. Из-за этого конфигурирование системы несколько усложняется и для такого способа настройки требует немного больше времени, но появляется возможность указывать, какие модули загружать не нужно. Преимущество этого подхода состояло, прежде всего, в экономии памяти, используемой приложением. Однако это также хорошо для соблюдения некоторого уровня безопасности. Лучше не загружать модуль, у которого есть возможность осуществлять соединения в сети, в том случае, если этот модуль в действительности не нужен.

После того, как модули будут загружены, в базовой части Asterisk происходит регистрация всех компонентных абстракций, реализованных в этих модулях. Есть большое количество типов интерфейсов, которые можно реализовать внутри модулей и зарегистрировать в базовой части Asterisk. С помощью модуля можно регистрировать столько различных интерфейсов, сколько необходимо. Обычно в один модуль группируются интерфейсы, функционально связанные друг с другом.

1.2.1. Драйверы каналов

Из всех интерфейсов, имеющихся в системе Asterisk, интерфейс драйверов каналов (`channel driver`) является наиболее сложным и наиболее важным. В API каналов системы Asterisk предоставляется абстракция телефонного протокола, что позволяет пользоваться всеми другими возможностями системы Asterisk независимо от того, какой конкретно используется телефонный протокол. На данный компонент возлагается обязанность осуществлять преобразование между абстракцией канала Asterisk и конкретными особенностями телефонной технологии, реализация которой скрыта с помощью этой абстракции.

Интерфейсом драйверов каналов Asterisk является интерфейс `ast_channel_tech`. В нем определяется набор методов, которые должны быть реализованы в драйвере канала. Первым методом, который должен быть реализован в драйвере канала, является метод фабрики `ast_channel`, который представляет собой метод `requester` объекта `ast_channel_tech`. Когда создается канал Asterisk для входящего или исходящего телефонного вызова, то с типом канала ассоциируется реализация объекта `ast_channel_tech`, который для этого вызова должен создавать конкретный экземпляр объекта `ast_channel` и осуществлять его инициализацию.

Когда объект `ast_channel` создан, он будет ссылаться на объект `ast_channel_tech`, с помощью которого он был создан. Есть много других операций, которые должны обрабатываться вполне конкретным способом, зависящим от технологических особенностей. Когда в канале `ast_channel` должны выполняться такие операции, то их выполнение происходит через обращение к соответствующему методу в объекте `ast_channel_tech`. На рис.1.2 показаны два канала в системе Asterisk. На рис.1.4 видно, как уровни абстракции и конкретные реализации вписываются в архитектуру Asterisk.

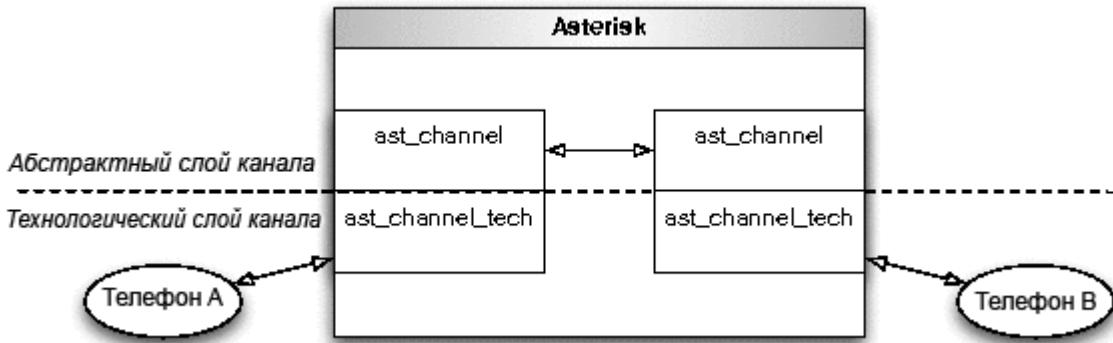


Рис.1.4: Слой технологической реализации канала и абстрактный слой канала

Наиболее важными методами в `ast_channel_tech` являются следующие:

- `requester`: Эта функция обратного вызова используется для запроса драйвера канала для создания экземпляра объекта `ast_channel` и инициализировать его так, как это требуется для канала этого типа.
- `call`: Эта функция обратного вызова используется для инициирования исходящего вызова к конечной точке, представленной в объекте `ast_channel`.
- `Answer`: Этот метод вызывается, когда система Asterisk решает, что должна ответить на входящий вызов, связанный с данным объектом `ast_channel`.
- `hangup`: Этот метод вызывается, когда система определяет, что нужно завершить вызов. После этого драйвер канала передаст конечному устройству сообщение в виде, соответствующему используемому протоколу, о том, что разговор завершен.
- `indicate`: Когда поступает вызов, то может произойти ряд событий, о которых нужно про-сигнализировать конечному устройству. Например, если устройство переводит вызов в режим удержания, то вызывается эта функция обратного вызова, которая сообщает о возникшем состоянии. Может быть применен метод, соответствующему используемому протоколу, который указывает, что вызов находится на удержании, либо драйвер канала может в ожидающем устройстве просто начать воспроизведение музыки.
- `send_digit_begin`: Эта функция вызывается с тем чтобы указать на начало набора тонального сигнала (DTMF) на устройстве по другую сторону канала.
- `send_digit_end`: Эта функция вызывается с тем чтобы указать на завершение набора тонального сигнала (DTMF) на устройстве по другую сторону канала.
- `read`: Эта функция вызывается базовой частью системы Asterisk с тем, чтобы повторить (`ast_frame`), полученный от этого конечного устройства. (`ast_frame`) является абстракцией в Asterisk, которая используется для инкапсуляции медиаданных (например, аудио или видео данных), а также для сигнализации о событиях.
- `write`: Эта функция используется для отправки фрейма `ast_frame` к данному устройству. Драйвер канала возьмет данные и упакует их согласно телефонному протоколу, который реализован и используется для передачи данных в конечное устройство.
- `Bridge`: Это функция обратного вызова нативного моста для канала данного типа. Как уже говорилось ранее, создание нативного моста происходит, когда драйвер канала может реализовать более эффективный мост для двух каналов одного и того же типа, а не направлять все сигналы и медиаданные через дополнительные ненужные слои абстракции. Это исключительно важно для повышения производительности.

После того, как вызов завершится, код абстрактного канала, который работает в базовой части Asterisk, обратится к функции обратного вызова `ast_channel_tech hangup` и уничтожит объект `ast_channel`.

1.2.2. Приложения планирования прохождения вызовов (dialplan)

Администраторы системы Asterisk настраивают маршруты прохождения вызовов с помощью плана Asterisk dialplan, который находится в файле `/etc/asterisk/extensions.conf`. Dialplan состоит из наборов правил, которые называются расширениями (extensions). Когда в систему поступает новый телефонный вызов, набранный номер используется для того, чтобы найти в dialplan-е расширение, которое должно использоваться для обработки данного вызова. В расширении указывается список приложений dialplan, которые должны быть выполнены для данного канала. Приложения, доступные для выполнения в dialplan-е, регистрируются в реестре приложений. Этот реестр заполняется во время загрузки модулей в систему.

В состав Asterisk входит почти две сотни приложений. Определение приложения очень свободное. В приложениях для взаимодействия с каналами могут использоваться любые внутренние интерфейсы API, имеющиеся в системе Asterisk. Некоторые приложения выполняют одну задачу, например, приложение `Playback`, которое для вызывающего проигрывает звуковой файл. Другие приложения вовлечены в работу системы в большей степени и выполняют очень большое количество операций, как, например, приложение `Voicemail`.

Благодаря тому, что используется Asterisk dialplan, несколько приложений можно объединять вместе для того, чтобы можно было выполнять специальную обработку вызовов. На случай, когда необходима более специализированная настройка, выходящая за рамки возможностей языка, предоставляемого в dialplan, есть скриптовые интерфейсы, которые позволяют выполнять специализированную обработку вызовов с использованием любого языка программирования. Даже в тех случаях, когда используются такие скриптовые интерфейсы с другими языками программирования, можно при взаимодействии с каналом продолжать обращаться к приложениям dialplan.

Прежде, чем мы перейдем к примеру, давайте посмотрим на синтаксис Asterisk dialplan, в котором обрабатываются вызовы с номером 1234. Заметьте, что номер 1234 выбран произвольным образом. Здесь вызываются три приложения dialplan. Первое, отвечающее на вызов. Следующее, проигрывающее звуковой файл. Заключительное, завершающее вызов.

```
; Определение правил, используемых в случае набора номера 1234.
;
exten => 1234,1,Answer()
    same => n,Playback(demo-congrats)
    same => n,Hangup()
```

Ключевое слово `exten` используется для определения расширения. В правой части строки `exten` цифры 1234 означают, что мы определяем правила для случая набора номера 1234. Следующая 1 означает, что это первый шаг, который нужно сделать, когда будет набран данный номер. Наконец, `Answer` указывает, что система должна ответить на вызов. Следующие две строки, которые начинаются с ключевого слова `same`, являются правилами для последнего расширения, для которого было задано определения, в данном случае — для 1234. Если кратко, то `n` означает, что делается следующий шаг. В последнем элементе каждой из этих строк, определяется, какое действие должно быть выполнено.

Ниже приведен еще один пример использования Asterisk dialplan. В этом случае система ответит на входящий вызов. Будет воспроизведен звуковой сигнал (beep), а затем в переменной `DIGITS` будет запомнено до четырех нажатых клавиш. Затем запомненные значения клавиш будут прочитаны и переданы вызывающей стороне. Наконец, вызов будет завершен.

```
exten => 5678,1,Answer()
    same => n,Read(DIGITS,beep,4)
    same => n,SayDigits(${DIGITS})
    same => n,Hangup()
```

Как уже упоминалось ранее, определение приложения является весьма свободным – зарегистрированный прототип функции очень простой:

```
int (*execute)(struct ast_channel *chan, const char *args);
```

Однако в действительности, в реализациях приложений могут использоваться все интерфейсы API, которые есть в `include/asterisk/`.

1.2.3. Функции dialplan

В большинстве приложений `dialplan` используется строка аргументов. Хотя некоторые из этих значений можно задать жестко, для того, чтобы обеспечить большую динамику поведения, вместо них можно пользоваться переменными. В следующем примере показан фрагмент `dialplan`, в котором устанавливается значение переменной, а затем с помощью приложения `Verbose` её значение выдается в командную строку Asterisk.

```
exten => 1234,1,Set(MY_VARIABLE=foo)
same => n,Verbose(MY_VARIABLE is ${MY_VARIABLE})
```

Функции `dialplan` вызываются с использованием точно такого же синтаксиса, как в предыдущем примере. Модули Asterisk могут регистрировать функции `dialplan`, с помощью которых будет собираться некоторая информация и передаваться в `dialplan`. Либо, наоборот, с помощью этих функций можно извлекать информацию `dialplan` и выполнять действия с учетом этой информации. Хотя функции `dialplan` могут устанавливать или получать мета данные, используемые каналом, они, как правило, не выдают никаких сигналов и не выполняют обработку медиа контента. Эта работа возлагается на приложения `dialplan`.

В следующем примере демонстрируется использование функции `dialplan`. Во-первых, в командную строку Asterisk выводится идентификатор `CallerID` текущего канала. Затем, с помощью приложения `SET` изменяется значение `CallerID`. В этом примере, `Verbose` и `SET` являются приложениями маршрутизации, а `CALLERID` является функцией.

```
exten => 1234,1,Verbose(The current CallerID is ${CALLERID(num)})
same => n,Set(CALLERID(num)=<256>555-1212)
```

В данном случае функция `dialplan` используется вместо обычной переменной, поскольку информация `CallerID` хранится в структуре данных в экземпляре `ast_channel`. В коде функции `dialplan` известно, как этой структуре присваивать значения и как из нее значения извлекать.

В следующем примере использования функции `dialplan` показано, как добавлять специальную информацию в журналы вызовов, называемыми записями CDR (Call Detail Records или подробные записи о вызове). Функция `CDR` позволяет получать записи с подробной информацией о вызове, а также добавлять специальную информацию.

```
exten => 555,1,Verbose(Time this call started: ${CDR(start)})
same => n,Set(CDR(mycustomfield)=snickerdoodle)
```

1.2.4. Трансляторы кодеков

В мире VOIP используется много различных кодеков, применяемых для кодирования медиа информации, пересыпаемой по сети. Разнообразие вариантов обусловлено компромиссами, касающимися качества передаваемой медиа информации, уровня загрузки процессора и требований, связанных с пропускной способностью. В системе Asterisk поддерживается множество различных кодеков и известно, как, при необходимости, осуществлять между ними преобразование.

Когда устанавливается соединение, Asterisk для того, чтобы не надо было делать преобразований, будет пытаться сделать так, чтобы на двух конечных устройствах использовались одинаковые медиакодеки. Однако, это возможно не всегда. Даже если используются одинаковые кодеки, все равно может потребоваться преобразование. Например, если система Asterisk настроена так, что когда

аудиосигнал проходит через систему, требуется какая-нибудь его обработка (например увеличение или уменьшение уровня громкости), то системе Asterisk прежде, чем она сможет выполнить обработку сигнала, потребуется перекодировать аудио сигнал в распакованные варианты. Система Asterisk также может быть сконфигурирована так, чтобы записывать телефонные вызовы. Если указанный в конфигурации формат записи отличается от того, который используется при вызове, то потребуется перекодировка.

Взаимодействие кодеков

То, как осуществляется выбор, какой кодек будет использоваться для медиа-потока, конкретно зависит от технологии, используемой для передачи вызова в систему Asterisk. В некоторых случаях, например, как вызов в традиционной телефонной сети (PSTN), вариантов выбора вообще нет. Однако в других случаях, особенно при использовании протоколов IP, выбирается такой способ, когда для обеих сторон указываются возможности и предпочтения и выбирается общий подходящий кодек.

Например, когда вызов посыпается в систему Asterisk, то, в случае использования протокола SIP (протокол, наиболее часто используемый в сетях VOIP), процедура выбора кодека, в самом общем виде, будет выглядеть следующим образом:

1. Конечное устройство посылает в систему Asterisk запрос на новый вызов с указанием списка кодеков, которые желательно использовать.
2. Asterisk проверяет свой конфигурационный файл, настроенный администратором, в котором в порядке предпочтения указывается список кодеков, разрешенных к использованию. Asterisk выберет наиболее предпочтительный кодек (на основе своего собственного заранее сконфигурированного предпочтения), который указан в конфигурационном списке конфигурации Asterisk и который также указан как поддерживаемый в поступившем запросе.

Одна из областей, с которой Asterisk справляется недостаточно хорошо, является область использования более сложных кодеков, особенно видеокодеков. За последние десять лет процедура согласование кодеков существенно усложнилась. Нам нужно многое сделать с тем, чтобы можно было пользоваться новейшими аудио кодеками и чтобы была возможность поддерживать видеокодеки гораздо лучше, чем мы это делаем сегодня. Это один из главных приоритетов новых разработок для следующего крупного релиза системы Asterisk.

В модулях трансляторов кодеков предлагается одна или несколько реализаций интерфейса `ast_translator`. В трансляторе есть атрибуты исходного и целевого форматов. Также имеется функция обратного вызова, которая будет использоваться для преобразования части медиапотока из исходного формата в целевой. В ней совсем ничего неизвестно о концепции телефонного вызова. В этой функции только известно, как конвертировать мультимедийные данные из одного формата в другой.

Более подробную информацию об интерфейсе API трансляторов кодеков смотрите в `include/asterisk/translate.h` и в `main/translate.c`. Реализации абстракции трансляторов можно найти в каталоге `codecs`.

1.3. Потоки

Система Asterisk является очень «тяжелым» многопоточным приложением. Оно использует интерфейс API потоков POSIX для управления потоками и относящимися к ним сервисами, например, блокировками. Весь код системы Asterisk, взаимодействующий с потоками, сделан так, что в нем присутствует набор оберток, используемых для отладки. Большую часть потоков в Asterisk можно считать либо потоком мониторинга сети (Network Monitor Thread), либо канальным потоком (Channel Thread), который иногда называется потоком АТС (автоматической телефон-

ной станции), поскольку его первоначальное назначение состояло в выполнении в канале функции АТС.

1.3.1. Потоки мониторинга сети

Потоки мониторинга сети имеются в системе Asterisk в каждом из основных драйверов канала. На них возлагается обязанность вести наблюдение за сетью, к которой они подключены (независимо от того, будет ли это сеть IP или обычная телефонная сеть), а также за всеми поступающими вызовами или другими типами поступающих запросов. С их помощью также осуществляется первоначальная настройка соединения, например, аутентификация и проверка набранного номера. После того, как настройка вызова будет выполнена, потоки мониторинга сети создадут экземпляр канала (`ast_channel`) и запустят канальный поток, которые будет обрабатывать вызов в течение всего оставшегося времени существования вызова.

1.3.2. Канальные потоки

Как уже обсуждалось ранее, канал является фундаментальной концепцией в системе Asterisk. Каналы могут быть входящими или исходящими. Входящий канал создается тогда, когда в систему Asterisk поступает входящий вызов. Эти каналы являются именно теми, которые выполняются согласно описаниям `dialplan` системы Asterisk. Для каждого канала, который выполняется согласно `dialplan`, создается поток. Такие потоки называются канальными потоками.

Приложения `dialplan` всегда исполняются в контексте канального потока. Функции `dialplan` почти всегда исполняются таким же образом. С помощью асинхронного интерфейса, например, интерфейса командной строки Asterisk, можно читать и записывать функции `dialplan`. Тем не менее, всегда есть канальный поток, который является владельцем структуры данных `ast_channel` и управляет временем жизни объекта.

1.4. Сценарии вызовов

В двух предыдущих разделах мы рассмотрели важные интерфейсы компонентов системы Asterisk, а также ее потоковую модель выполнения. В этом разделе для того, чтобы продемонстрировать, как компоненты системы Asterisk компоненты совместно обрабатывают телефонные вызовы, будут подробно рассмотрены несколько наиболее распространенных сценариев вызовов.

1.4.1. Проверка голосовой почты

Одним из типичных сценариев является ситуация, когда некто звонит для того, чтобы проверить свою голосовую почту. Первым компонентом, которые участвует в этом сценарии, является драйвер канала. Драйвер будет ответственен за обработку входящего вызова, который появится в потоке мониторинга в драйвере канала. Это вызов может настраиваться по-разному в зависимости от технологии, используемой для пересылки вызова в систему. Следующим шагом настройки вызова является определение, кому отправляется вызов. Это обычно осуществляется по номеру, который набираетзывающий. Однако, в некоторых случаях в наличии нет никакого конкретного номера, поскольку в технологии, используемой для пересылки вызова, нет спецификации, связанной с набранным номером. Примером этого может быть входящий вызов, поступающий по аналоговой телефонной линии.

Если драйвер канала определит, что для набранного номера в конфигурации Asterisk заданы расширения, определенные в `dialplan`-е (конфигурация маршрутизации вызова), то он создаст объект канала Asterisk (`ast_channel`) и создаст канальный поток. На этот канальный поток будет возложена основная обязанность обработки остальной части вызова (рис.1.5).

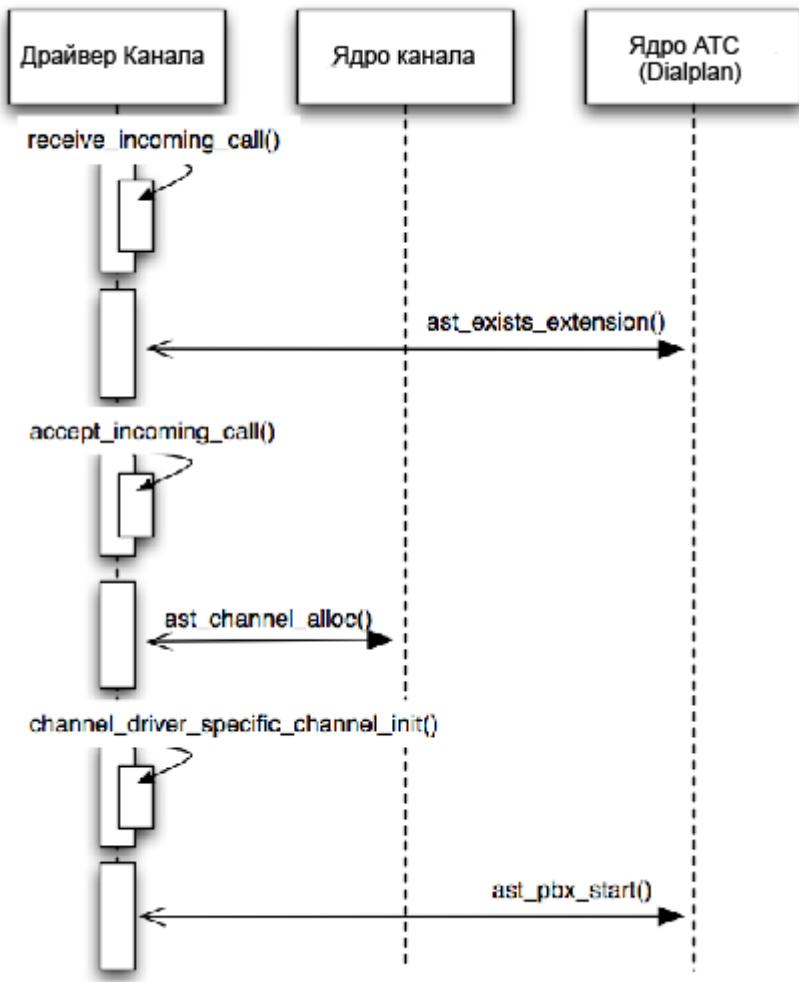


Рис.1.5.: Диаграмма последовательности настройки вызова

Главный цикл канального потока осуществляет выполнение dialplan-а. Он следует правилам, определенным для набранного расширения, и выполняет шаги, которые в них определены. Ниже приведен пример расширения, записанного в синтаксисе extensions.conf. В этом расширении указывается, что нужно ответить на вызов и, если кто-нибудь наберет номер *123, выполнить приложение VoicemailMain. Это то приложение, которое вызывается пользователем для проверки сообщений, оставшихся в голосовой почте.

```

exten => *123,1,Answer()
        same => n,VoicemailMain()

```

Когда канальный поток выполнит приложение Answer, Asterisk ответит на входящий вызов. Ответ на вызов требует специальной технологической обработки, поэтому в добавок к некоторой общей обработке ответа также вызывается функция обратного вызова answer, находящаяся в ассоциированной структуре ast_channel\tech, которая будет обрабатывать вызов. При этом может происходить отсылка по сети специального пакета поверх сети IP, сообщающего, что в аналоговой линии нужно снять трубку и т.д.

Следующим шагом для канального потока будет выполнение приложения VoicemailMain (рис.1.6). Это приложение предоставляется в виде модуля app_voicemail. Следует заметить, что хотя приложение Voicemail выполняет большой объем работы, необходимый при взаимодействии с вызовами, в нем ничего не известно о технологии, используемой для передачи вызова в систему Asterisk. Абстракция канала The Asterisk скрывает эти особенности от реализации приложения голосовой почты.

Есть целый ряд возможностей, которые предоставляются обратившемуся к своей голосовой почте. Однако все они, прежде всего, реализованы как операции чтения и записи звуковых файлов, выполняемые в ответ на ввод команд, поступающих от звонящего, которые, в основном, являются нажатием клавиш с цифрами. Сигналы о нажатии клавиш могут поступать в Asterisk различными способами. Но опять, с этими особенностями справляются драйверы каналов. Как только нажатие клавиши поступает в систему Asterisk, оно конвертируется в обобщенное событие о нажатии клавиши и передается в код приложения Voicemail.

Одним из важных интерфейсов в системе Asterisk, который мы уже рассматривали, является транслятор кодеков. Такие реализации кодеков важны для данного сценария вызова. Когда в коде голосовой почты потребуется вызывающему воспроизвести звуковой файл, аудиоформат в звуковом файле может не совпасть с аудиоформатом, используемым при коммуникации между системой Asterisk и вызывающим. Если потребуется преобразовать аудиоформат, то для того, чтобы из исходного формата получить целевой формат, будет создан последовательность преобразований (translation path), состоящая из одного или нескольких трансляторов кодеков.

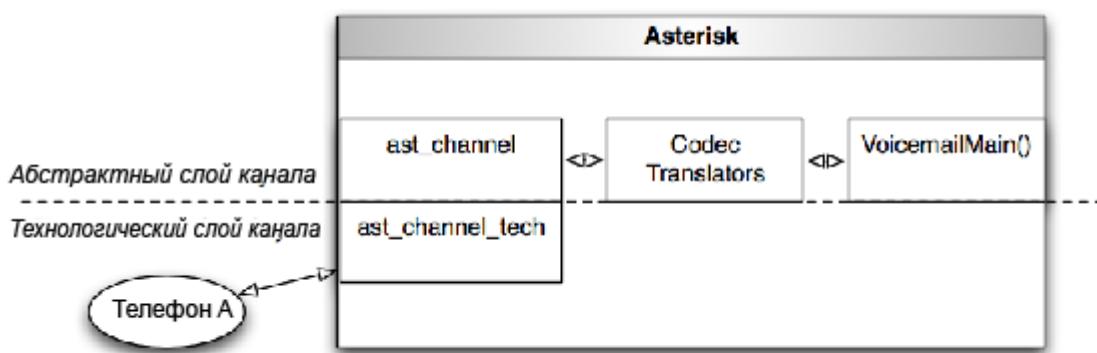


Рис.1.6: Вызов приложения голосовой почты VoicemailMain

В некоторый момент вызывающий завершит свое общение с системой голосовой почты и повесит трубку. Драйвер канала обнаружит, что это произошло, и преобразует это событие в обобщенное сигнальное событие канала Asterisk. Код приложения Voicemail получит это сигнальное событие и закончит свое выполнение. Управление будет возвращено в основной цикл в канальном потоке, в результате чего можно будет продолжить выполнение dialplan-a. Поскольку в этом примере в dialplan-e нет ничего, что нужно обрабатывать, драйвер канала получит возможность специальный образом, зависящим от используемой технологии, обработать шаг отключения вызова, после чего объект `ast_channel` будет уничтожен.

1.4.2. Вызов с созданием «моста»

Еще одним довольно общим сценарием вызовов в системе Asterisk является вызов с созданием «моста» между двумя каналами. Это сценарий, когда один телефон через систему вызывает другой телефон. Первоначальный этап процесса настройки вызова аналогичен тому, который был рассмотрен в предыдущем примере. Отличия в обработке начинаются после того, как вызов был настроен и канальный поток начинает выполнение dialplan-a.

Следующий dialplan является простым примером, результатом выполнения которого будет вызов с использованием соединения типа «мост». Когда на телефоне будет набран номер 1234, то при использовании данного расширения dialplan выполнит приложение `Dial`, которое является основным приложением, используемым для инициации исходящего вызова.

```
exten => 1234,1,Dial(SIP/bob)
```

Аргумент, указываемый в приложении `Dial`, сообщает, что система должна создать исходящий вызов на устройство, обозначаемое как `SIP/bob`. Часть `SIP` этого аргумента указывает, что для доставки данного вызова должен использоваться протокол SIP. `bob` будет проинтерпретировано драйвером канала, который должен реализовывать протокол SIP, т.е. `chan_sip`. Если предположить, что в драйвере канала был должным образом сконфигурирован аккаунт с именем `bob`, то будет известно, как можно будет получить доступ к телефону Боба.

Приложение `Dial` запросит базовую часть системы Asterisk выделить новый канал, использующий идентификатор `SIP/bob`. Базовая часть запросит драйвер канала SIP выполнить специальную инициализацию, соответствующую данной технологии. Драйвер канала также инициирует процесс звона до телефона. Как только придет ответ на этот вызов, драйвер передаст события обратно в базовую часть Asterisk, которые будут приняты приложением `Dial`. Эти события могут говорить о том, что поступил ответ на вызов, что направление занято, что сеть перегружена, что вызов был отклонен по некоторой причине или о ряде других событий. В идеальном случае будет получен ответ на вызов. Тот факт, что на вызов был получен ответ, вернется обратно во входящий канал. Asterisk не ответит на входящий вызов до тех пор, пока не на исходящий вызов не поступит ответ. Ка только оба канала ответят на вызов, между ними будет установлено соединение типа «мост» (рис.1.7).

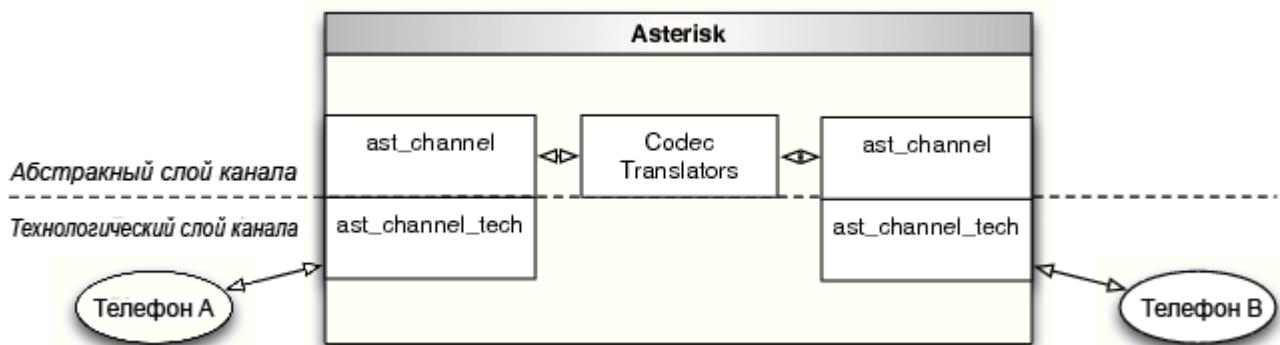


Рис.1.7: Блок-схема мостового вызова внутри обобщенной абстракции типа «мост»

Когда через мост произойдет соединения каналов, аудиопоток и сигнальные события будут передаваться между каналами до тех пор, пока не возникнет некоторое событие, ведущее к разрыву мостового соединения, например, когда на одной стороне повесят трубку. На рис.1.8 приведена диаграмма, демонстрирующая ключевые операции, выполняемые с аудиофреймом при использовании соединения типа «мост».

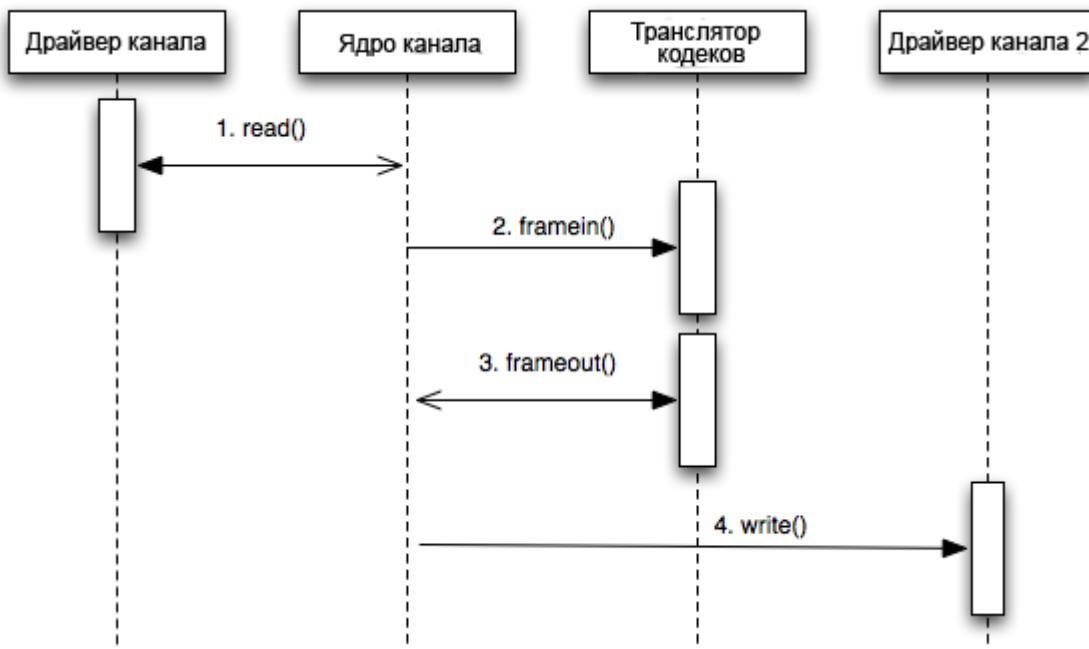


Рис.1.8: Диаграмма последовательности действий при обработке аудио фрейма для соединения типа «мост»

После того, как вызов будет завершен, процесс рассоединения будет похож на тот, что описан в предыдущем примере. Главное отличие здесь в том, что данном процессе участвуют два канала. Прежде, чем канальный поток прекратит свое существование, для обоих каналов будет выполнена специальная процедура рассоединения, обусловленная технологией, используемой в каждом из каналов.

1.5. Заключительные комментарии

Архитектуре системы Asterisk в настоящее время уже больше десяти лет. Однако фундаментальные концепции каналов и гибкой обработки вызовов, используемых в dialplan-е системы Asterisk, по-прежнему позволяют поддерживать разработку сложных систем телефонии в отрасли, которая постоянно развивается. Одной из областей, с которой архитектура системы Asterisk не может достаточно хорошо справиться, является масштабирование системы на нескольких серверах. Сообщество разработки Asterisk разрабатывает в настоящее время сопутствующий проект, который называется Asterisk SCF (Scalable Communications Framework — Масштабируемый коммуникационный фреймворк), предназначенный для решения этих проблем масштабируемости. В ближайшие несколько лет мы ожидаем увидеть, что система Asterisk вместе с системой Asterisk SCF будет продолжать занимать все большую часть рынка телефонии, в том числе в области крупных систем.

Примечания

1. <http://www.asterisk.org/>
2. DTMF является сокращением Dual-Tone Multi-Frequency (режим двухтонального многочастотного набора). Это тональный сигнал, который посыпается как телефонный аудиосигнал, когда кто-нибудь на своем телефоне нажимает клавишу с цифрой.

2.Audacity

Audacity является популярной программой записи звука и популярным аудио-редактором. Эта программа достаточно мощная и, в то же время, проста в использовании. Большинство ее пользователей работают на Windows, но тот же самый исходный код Audacity можно откомпилировать для использования также на Linux и Mac.

Доминик Мазони (Dominic Mazzoni) написал первую версию Audacity в 1999 году, когда он был аспирантом Университета Карнеги-Меллона. Доминик хотел создать платформу для разработки и отладки алгоритмов обработки аудиозаписей. Программа развивалась и расширялась сферы ее применения. Как только Audacity была выпущена с открытым исходным кодом, она привлекла других разработчиков. На протяжении многих лет небольшая не сильно меняющаяся команда энтузиастов модифицировала, сопровождала, обновляла эту программу, писала для нее документацию, помогала пользователям и переводила интерфейс Audacity на другие языки.

Одна из целей создания этой программы состояла в том, что сразу можно обнаружить в ее пользовательском интерфейсе: у людей должна быть возможность сесть и сразу без всяких руководств начать пользоваться программой, постепенно открывая для себя новые возможности. Этот принцип оказал гораздо большее влияние на согласованность пользовательского интерфейса Audacity, чем что-либо другое. Для проекта, который у многих всегда под рукой, такой принцип унификации важнее, чем это кажется на первый взгляд.

Было бы хорошо, если архитектура Audacity следовала бы такому же принципу, которому придерживаются в пользовательском интерфейсе. Лучшее, что у нас для этого есть, это - «попробовать и быть последовательными». Когда добавляется новый код, разработчики стараются следовать стилю и соглашениям, которые есть в коде, расположенному по соседству. Однако, на практике база кода Audacity представляет собой смесь хорошо структурированного кода и кода, который структурирован несколько хуже. Скорее всего, вся архитектура в общем похожа на небольшой город: есть несколько впечатляющих зданий, но вы также найдете и захудальные районы, которые больше напоминают трущобы.

2.1. Структура Audacity

Audacity состоит из слоев нескольких библиотек. Хотя для большей части нового кода, который программируется в Audacity, не требуется детального знания того, что именно происходит в этих библиотеках, знакомство с их интерфейсом API, и то, что они делают, является важным. Двумя наиболее важными библиотеками являются библиотека PortAudio, в которой предоставлен низкоуровневый аудио интерфейс, обеспечивающий кросс-платформенность, и библиотека wxWidgets, в которой предоставлен графический компонент кросс-платформенности.

Когда читаете код Audacity, полезно понимать, что важной является только часть кода. Библиотеки добавляют много необязательных функций, хотя те, кто ими пользуется, не считают, что они необязательные. Например, имея свои собственные встроенные звуковые эффекты, Audacity поддерживает LADSPA (Linux Audio Developer's Simple Plugin API — интерфейс API простых плагинов разработчиков Linux Аудио) для динамически загружаемых плагинов звуковых эффектов. Интерфейс VAMP API в Audacity делает то же самое для плагинов, которые анализируют аудио. Без этих интерфейсов Audacity была бы менее функционально насыщенной, но она абсолютно не зависит от этих возможностей.

Другими необязательными библиотеками, используемыми Audacity, являются libFLAC, libogg и libvorbis. С их помощью реализуются различные форматы сжатия аудиосигнала. Формат MP3 реализуется за счет динамической загрузки библиотеки LAME или Ffmpeg. Лицензионные ограничения не позволяют встраивать эти очень популярные библиотеки сжатия.

Лицензирование стоит за некоторыми другими решениями, относящимся к библиотекам и структурам Audacity. Например, поддержка плагинов VST не встраивается из-за лицензионных ограни-

чений. Нам бы также хотелось в некоторых местах нашего использовать очень эффективный код FFTW , реализующий быстрое преобразование Фурье. Однако, мы предоставляем это только как дополнительный вариант для тех, кто компилирует Audacity самостоятельно, а вместо этого возвращаемся к версии, которая встроена в наш код и работает чуть медленнее. До тех пор, пока в Audacity можно пользоваться плагинами, можно и следует утверждать, что в Audacity не должен использоваться пакет FFTW. Авторы FFTW не хотят, чтобы их код был доступен в виде обычного сервиса в каком-либо другом коде. Так, архитектурное решение о поддержке плагинов приводит к компромиссу, относящемуся к тому, что мы можем предложить. Он позволяет в наших готовых исполняемых файлах использовать плагины LADSPA, но запрещает нам использовать FFTW.

На архитектуре сказались также и наши мысли о том, как лучше использовать наше ограниченное время, уделяемое разработке. С небольшой командой разработчиков, у нас нет, например, ресурсов, чтобы произвести углубленный анализ лазеек, связанных с безопасностью, который делают группы, работающие над Firefox и Thunderbird. Тем не менее, мы не хотим, чтобы Audacity было средством, позволяющим обходить брандмауэр, поэтому у нас есть правило вообще не иметь в Audacity входящих или исходящих соединений TCP/IP. Отсутствие соединений TCP/IP позволяет избежать многих проблем безопасности. Наше понимание того, что мы обладаем ограниченными ресурсами, ведет нас к лучшим проектным решениям. Это помогает нам отказываться от функций, на разработку которых нам придется тратить слишком много времени, и сосредотачиваться на том, что наиболее важно.

Аналогичную озабоченность, касающуюся времени разработки, связана со скриптовыми языками. Мы хотим писать сценарии, но код, реализующий эти языки, не должен быть в Audacity. Нет смысла создавать в Audacity копии всех скриптовых языков лишь для того, чтобы дать пользователям возможность выбрать тот, который им нужен [1]. Вместо этого нами реализована возможность писать скрипты с помощью единственного модуля плагина и конвейера, что мы рассмотрим позже.

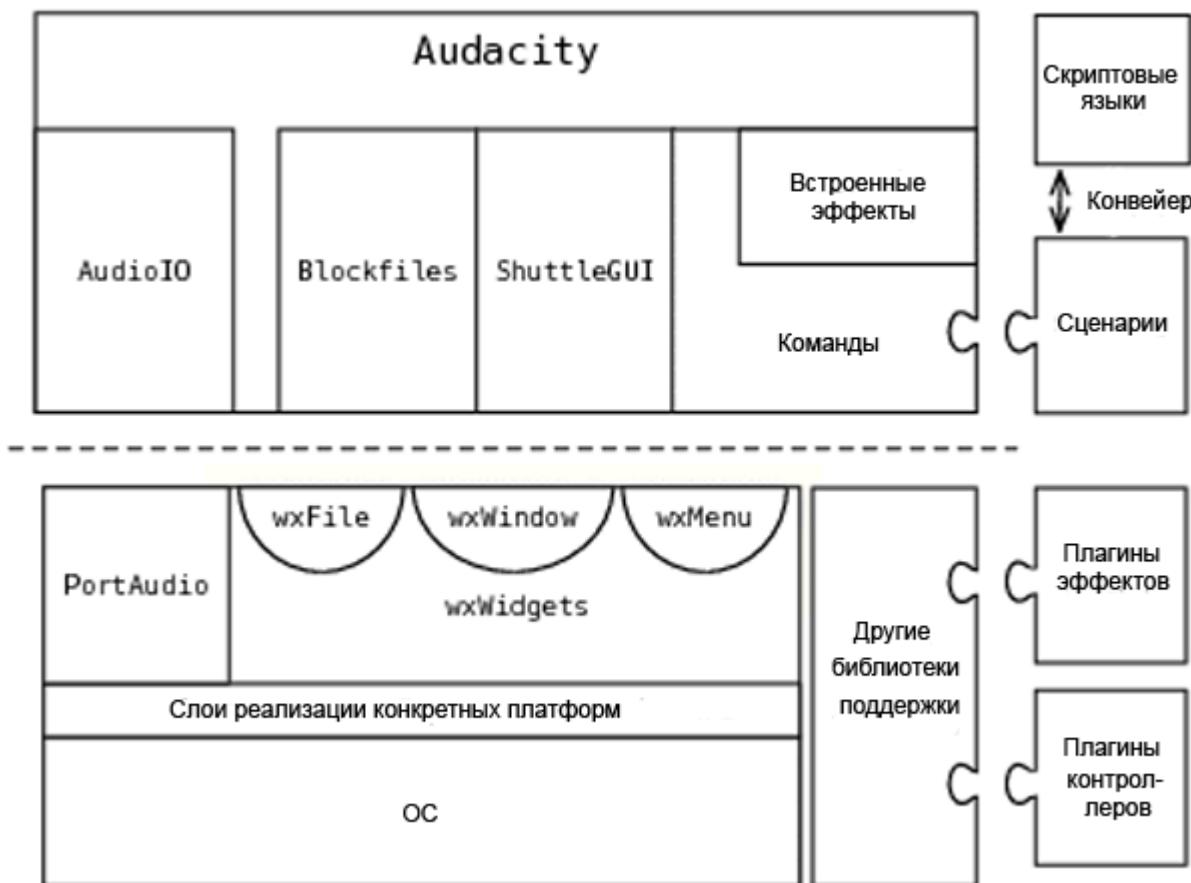


Рис.2.1: Слои в Audacity

На рис.2.1 показаны несколько слоев и модулей, имеющиеся в Audacity. На схеме в wxWidgets выделены три важных классах, играющих важную роль в Audacity. Мы, опираясь на абстракциях низкого уровня, строим абстракции более высокого уровня. Например, система блок-файлов BlockFile отображается в объекты wxFile, имеющиеся в wxWidgets. Возможно, на некотором этапе, имеет смысл выделить блок-файлы BlockFile, графическую оболочку ShuttleGUI и команды обработки в промежуточную библиотеку с ее собственными правами. Это должно стимулировать нас сделать их более обобщенными.

Ниже в диаграмме показана узкая полоска «Слои реализации конкретных платформ». Оба слоя wxWidgets и PortAudio являются слоями абстракции ОС. В них обоих есть код, позволяющий, в зависимости от целевой платформы, делать выбор между различными реализациями.

В категорию «Другие библиотеки поддержки» включен широкий набор библиотек. Достаточно интересно то, что в многих из них используются динамически загружаемые модули. Эти динамические модули ничего не знают о wxWidgets.

На платформе Windows мы, как правило, компилируем Audacity в один монолитный исполняемый файл, причем код приложений wxWidgets и Audacity входит в тот же самый исполняемый файл. В 2008 году мы перешли на использование модульной структуры и используем wxWidgets как отдельную DLL. Это позволяет во время выполнения программы загружать дополнительные DLL с необязательными функциями только тогда, когда в этих DLL непосредственно используются возможности wxWidgets. Плагины, которые на схеме подключены выше пунктирной линии, могут использовать wxWidgets.

Решение об использовании DLL для wxWidgets имеет свои минусы. Размер дистрибутива теперь стал больше, частично из-за того, что в DLL есть много неиспользуемых функций, которые раньше были просто удалены. Загрузка Audacity также происходит дольше, поскольку каждая DLL загружается отдельно. Но преимущества больше. Мы надеемся, что модульность даст нам те же преимущества, какие есть в Apache. Мы видим, что модули позволяют ядру Apache быть очень стабильными, а поддержка экспериментов, специальных возможностей и новых идей происходит в модулях. Модули проходят очень долгий путь, противодействуя искушению изменить направление развития проекта. Мы думаем, что это для нас это было очень важным архитектурным решением. Мы ожидаем от этого выиграть, но еще мы этого не достигли. Предоставление функций wxWidgets в таком виде является лишь первым шагом, и мы должны многое сделать для того, чтобы получить более гибкую модульную систему.

Структура программы такой, как Audacity, заранее четко не разрабатывается. Она разрабатывается по ходу дела. По большому счету, архитектура, которая у нас есть, для нас подходит. Нам приходится сражаться с архитектурой, когда мы пытаемся добавлять новые возможности, которые затрагивают многие файлы с исходным кодом. Например, в настоящее время в Audacity стерео и моно дорожки обрабатываются специальным образом. Если бы вы хотели изменить Audacity так, чтобы обрабатывать объемный звук, то вам бы потребовалось вносить изменения во многие классы в Audacity.

За пределами стереозвука: история о GetLink

Audacity никогда не имела абстракции числа каналов. Вместо этой абстракции имелись ссылки на аудиоканалы. Есть функция `GetLink`, которая возвращает другой парный звуковой канал в паре, если их два, или возвращает NULL, если дорожка монофоническая. Код, в котором используется `GetLink`, обычно выглядит точно так, как если бы его первоначально написали для монофонического сигнала, а затем использовали проверку (`GetLink() != NULL`) с тем, чтобы обрабатывать стереосигнал. Я не уверен, что крд был написан именно так, но подозреваю, что так это и произошло. Нет циклического использования `GetLink` по всем каналам, находящихся в связанном списке. При рисовании, микшировании, чтении и записи — везде используется проверка случая обработки стереосигнала, а не обобщенный код, который может работать на n каналах, где n, сколько

рее всего, должно быть равно одному или двум. Чтобы перейти к более обобщенному коду, вам потребуется внести изменения приблизительно в 100 мест, где вызывается функция `GetLink`, изменив, по меньшей мере, 26 файлов.

Легко выяснить, что найти вызовы `GetLink` и внести необходимые изменения с тем, чтобы исправить эту «проблему», не так уж сложно, как это может показаться на первый взгляд. История с `GetLink` не о структурных изъянах, которые трудно исправить. Она, скорее всего, иллюстрирует то, как относительно небольшой изъян может переместиться в различные части кода, если этому не воспрепятствовать.

Если оглянуться назад, то было бы хорошо сделать функцию `GetLink` приватной и предложить итератор для перебора всех каналов в дорожке. Это позволило бы отказаться от большого количества специальных случаев кода, предназначенных для стереосигналов, и, в то же время, создать код, в котором список аудио каналов использовался бы независимо от того, как реализован список.

Более модульный проект, вероятно, позволит нам лучше скрыть внутреннюю структуру. Когда мы определяем и расширяем внешние интерфейсы API, нам требуется более внимательно изучать функции, которые нам предлагаются. В результате наше внимание сосредотачивается на абстракциях, а мы не хотим, чтобы они присутствовали во внешнем API.

2.2. Библиотека графического интерфейса wxWidgets

Наиболее важной отдельно библиотекой, которую программисты Audacity используют для создания графического пользовательского интерфейса, является библиотека wxWidgets, в которой предлагаются такие вещи, как кнопки, движки, флаги, обычные и диалоговые окна. В ней реализована большая часть кроссплатформенных функций визуализации. В библиотеке wxWidgets есть свой собственный класс строк `wxString`, в ней есть кроссплатформенные абстракции потоков, файловых систем и шрифтов, а также механизм для локализации для других языков, которыми мы пользуемся. Мы советуем тем, кто только что присоединился к разработке Audacity, загрузить wxWidgets, скомпилировать некоторые из примеров, которые поставляются с этой библиотекой, и поэкспериментировать с ними. Библиотека wxWidgets является сравнительно тонким слоем, построенным над базовыми объектами графического пользовательского интерфейса, предоставляемыми операционной системой.

Для создания сложных диалогов в wxWidgets предлагаются не только отдельные элементы, но также специальные механизмы управления размерами элементов и их положением (*sizers*). Они позволяют получить более красивый внешний вид, чем при использовании только абсолютно фиксированных позиций графических элементов. Если размер виджета изменен либо непосредственно пользователем, либо, скажем, из-за изменения размера шрифта, на другой, изменение расположения элементов в диалоговом окне произойдет очень естественно. Такие механизмы очень важны для кроссплатформенных приложений. Без них нам, возможно, приходилось бы пользоваться для каждой платформы специальными вариантами диалоговых окон.

Часто дизайн таких диалоговых окон помещается в файл ресурсов, который читается программой. Однако в Audacity мы строим диалоговые окна в программе исключительно в виде последовательности вызовов wxWidgets. В результате достигается максимальная гибкость: то есть, диалоговые окна, конкретное содержимое и поведение которых будет определяться кодом уровня приложения.

В свое время вы могли бы найти в Audacity места, где было ясно, что исходный код для создания графического пользовательского интерфейса был сгенерирован с использованием графических инструментов создания диалоговых окон. Эти средства помогли нам получить базовый проект. Со временем для того, чтобы добавить новые функции, базовый код был повсюду изменен, в резуль-

тате чего во многих местах новые диалоговые окна создавались с помощью копирования и модификации существующих кода с некоторыми изменениями, касающихся диалоговых окон.

После нескольких лет такой разработки мы обнаружили, что значительная часть исходного кода Audacity, прежде всего диалоговые окна для задания пользовательских настроек, состоит из запутанного и повторяющегося кода. Этот код, хотя он был прост с точки зрения того, что делал, следовать ему было удивительно трудно. Часть проблемы состояла в том, что порядок, в котором были построены диалоговые окна, был абсолютно произвольным: мелкие элементы объединялись в более крупные и, в конечном итоге, в полные диалоговые окна, но порядок, в котором элементы создавались в коде, не соответствовал (что и не требовалось) тому порядку, в котором элементы располагались на экране. Код был излишне большого размера и было много повторений. Был код, являющийся частью графического пользовательского интерфейса, который передавал данные из пользовательских настроек, запомненных на диске, в промежуточные переменные, код, который передавал значения промежуточных переменных в изображаемый графический пользовательский интерфейс, код, который передавал данные из изображаемого графического пользовательского интерфейса в промежуточные переменные, и код, который передавал промежуточные переменные в пользовательские настройки, запоминаемыми на диске. В коде был вставлен комментарий `//this is a mess` (// здесь беспорядок), но прошло достаточно много времени до того момента, когда с этим было что-то сделано.

2.3. Слой ShuttleGui

Решить проблему с запутанным кодом позволил новый класс, `ShuttleGui`, который существенно сократил число строк кода, необходимого для спецификации диалоговых окон, что сделало код более удобным для чтения. Класс `ShuttleGui` представляет собой дополнительный слой между библиотекой `wxWidgets` и `Audacity`. Его задача заключается в передаче между ними информации. Ниже приведен пример, который, с конца концов, становится элементами графического пользовательского интерфейса, изображенного на рис.2.2.

```
ShuttleGui S;
// GUI Structure
S.StartStatic("Some Title", ...);
{
    S.AddButton("Some Button", ...);
    S.TieCheckbox("Some Checkbox", ...);
}
S.EndStatic();
```

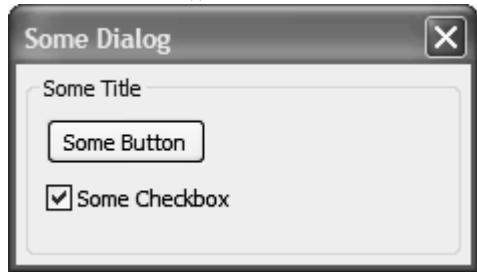


Рис.2.2: Пример диалогового окна

В этом коде определяется статическая область в диалоговом окне и область, в которой находится кнопка и чекбокс. Соответствие между кодом и диалоговым окном должно быть очевидным. `StartStatic` и `EndStatic` являются парными вызовами. Другими подобными согласованными параметрами являются `StartSomething/EndSomething`, которые используются для управления другими элементами компоновки диалогового окна. Фигурные скобки и отступы, которые с ними используются, требуются не для того, чтобы код был правильным. Мы приняли соглашение о том, чтобы их добавлять для того, чтобы сделать более очевидными структуру и особенности согласования парных вызовов. Это действительно упрощает чтение больших примеров.

Показанный исходный код не только создает диалоговое окно. Код, идущий после комментария "/* GUI Structure", также можно использовать для обмена данными между диалоговым окном и пользовательскими настройками, запоминаемыми на диске. Ранее для этого пришлось бы писать большой объем кода. В настоящее время этот код пишется только один раз, и скрыт в классе `ShuttleGui`.

В Audacity есть и другие расширения для базовых виджетов `wxWidgets`. В Audacity есть свой собственный класс для управления панелями инструментов. Почему не использовать класс панелей инструментов, встроенный в `wxWidget`? Причина историческая: панели инструментов Audacity были написаны до того, как стал использоваться `wxWidgets`, в котором предоставлен класс панелей инструментов.

2.4. Панель TrackPanel

Основной панелью в Audacity, на которой отображаются аудиодорожки, является панель `TrackPanel`. Это специальный управляющий компонент, который рисуется с помощью Audacity. Панель состоит из таких компонентов, как панели меньшего размера с информацией о дорожке, линейка для изменения развертки, линеек для изменения амплитуды, а также дорожек, на которых могут отображаться сигналы, спектры или текстовые метки. С помощью операции перетаскивания, выполняемой мышью, можно изменить размер и местоположение дорожки. Дорожки, у которых есть текстовые метки, пользуются нашей собственной реализацией редактируемого текстового поля, а не той, что есть в системе. Вы можете решить, эти дорожки и линейки, располагаемые панели, должны быть компонентами `wxWidgets`, но это неверно.

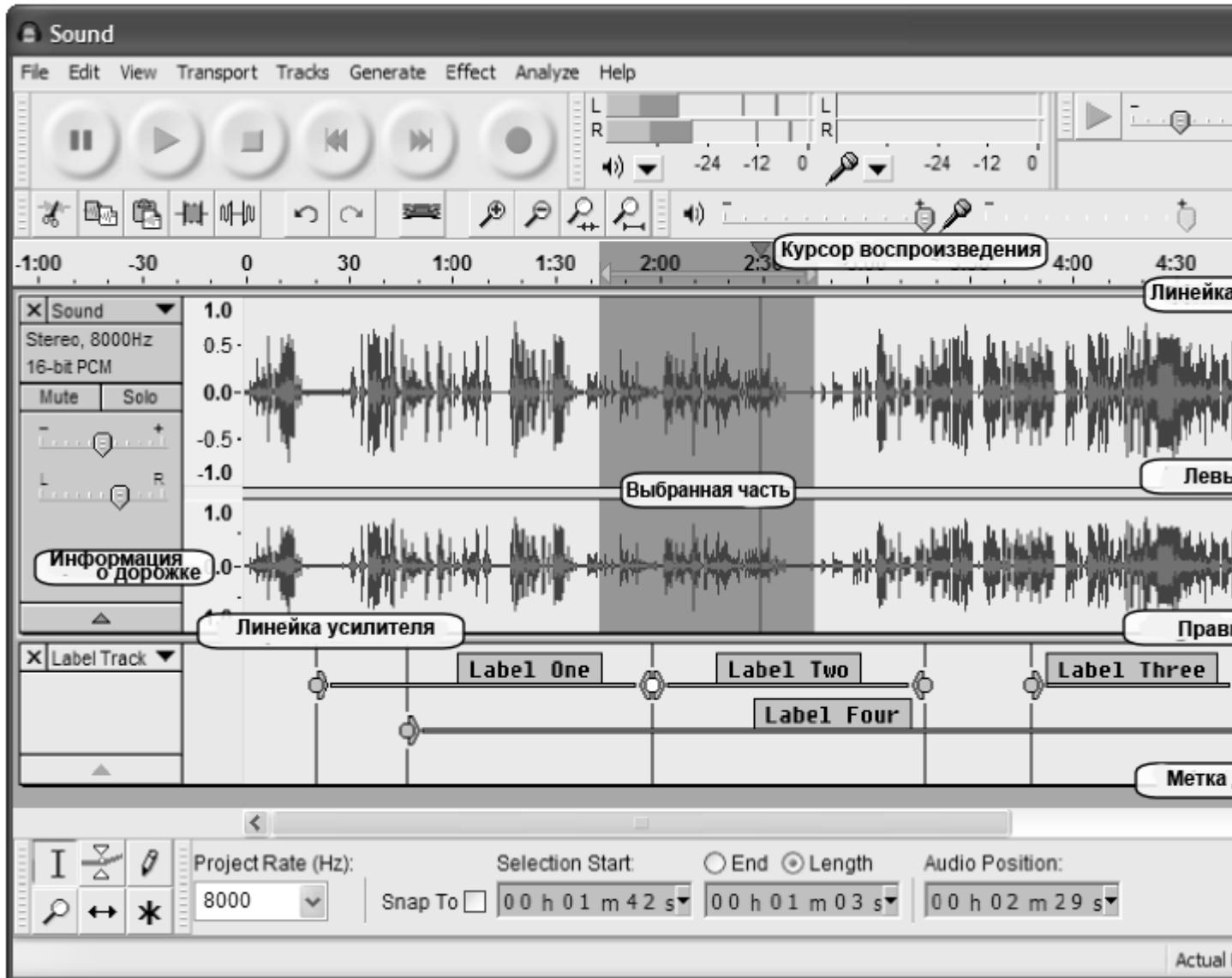


Рис.2.3: Интерфейс Audacity с элементами панели, текстовыми метками и аудиодорожками

На скриншоте, показанном на рис.2.3, приведен пользовательский интерфейс Audacity. Все компоненты, которые помечены, являются специальными компонентами Audacity. Что касается подключений wxWidgets, то в TrackPanel имеется один компонент wxWidget. Все позиционирование и перерисовку обеспечивает код Audacity, а не wxWidgets.

Способ, с помощью которого все эти компоненты собраны друг с другом для создания TrackPanel, действительно ужасен. Ужасен именно код; конечный результат, который пользователь видит, выглядит просто отлично. Код графического пользовательского интерфейса и код, относящийся конкретно к приложению, смешаны вместе, а не отделены друг от друга. В хорошем проекте только в коде приложения должно быть известно о левом и правом каналах, децибелах, отключение звука и режиме соло. Элементы графического интерфейса должны быть независимы от элементов приложения и должна быть возможность их повторного использования в приложении, не относящегося к аудиозаписям. Даже чисто графические части панели TrackPanel являются смешением специальных случаев кода, в котором указаны абсолютные позиции и размеры, и недостаточно абстракций. Было бы гораздо лучше, понятней и более естественный, если бы эти специальные компоненты были самодостаточными элементами графического пользовательского интерфейса и если они использовали значения размера точно также, как это делается в wxWidgets.

Чтобы получить такую панель TrackPanel, нам в wxWidgets нужен другой механизм использования значений размеров, который позволял перемещать и изменять размеры дорожек или, впрочем, любого другого виджета. В wxWidgets это делается недостаточно гибко. Мы бы могли использо-

вать этот механизм в других местах, что дало бы нам дополнительные преимущества. Мы могли бы пользоваться им в панелях инструментов, где находятся кнопки, что позволило бы более просто выбирать положения кнопок, перетаскивая их с места на место с помощью мыши.

Для этого была проделана определенная исследовательская работа, но в недостаточном объеме. Некоторые эксперименты с созданием графических компонентов полноценных элементов wxWidgets выявили следующую проблему: подобный механизм уменьшает возможность управлять перерисовкой виджетов, в результате возникает мерцание в тех случаях, когда изменяются размеры или происходит перемещение компонентов. Нам потребуется существенно изменить wxWidgets, чтобы избавиться от мерцания при перерисовке и еще лучше отделить операции, выполняемые при изменении размеров, от операций, необходимых при перерисовке.

Для этого была проделана определенная исследовательская работа, но в недостаточном объеме. Некоторые эксперименты с созданием графических компонентов полноценных элементов wxWidgets выявили следующую проблему: подобный механизм уменьшает возможность управлять перерисовкой виджетов, в результате возникает мерцание в тех случаях, когда изменяются размеры или происходит перемещение компонентов. Нам потребуется существенно изменить wxWidgets, чтобы избавиться от мерцания при перерисовке и еще лучше отделить операции, выполняемые при изменении размеров, от операций, необходимых при перерисовке.

Лучшим решением является использование простого шаблона, по которому мы самостоятельно нарисуем легковесные виджеты, что позволит не иметь соответствующих объектов, потребляющих ресурсы оконной системы, и не нужно будет ими управлять. Нам бы хотелось использовать структуру, подобную механизму выбора размеров, имеющуюся в wxWidgets, и виджеты компонентов, и использовать похожий интерфейс API, компоненты которого не являются производными от классов wxWidgets. Мы должны переделать наш существующий код TrackPanel так, чтобы его структура стала более ясной. Если бы это было простое решение, то это было бы уже сделано, но то, что мы пришли к единому мнению, что, в конечном итоге, мы хотим именно получить, позволило нам продвинуться дальше от предыдущих попыток. Если обобщить наш нынешний подход, то требуется выполнить большой объем работы по проектированию и кодированию. Есть большой соблазн отказаться от сложного кода, который уже и так работает достаточно хорошо.

2.5. Библиотека PortAudio: запись и воспроизведение

PortAudio является аудио-библиотекой, которая дает Audacity возможность воспроизводить и записывать звук независимо от используемой платформы. Без нее Audacity не сможет использовать звуковую карту устройства, на котором оно работает. В PortAudio предоставляются кольцевые буферы, средства, позволяющие изменять частоту дискретизации при воспроизведении/записи, и, самое главное, предоставляется интерфейс API, который скрывает различия между аудиообработкой на платформах Mac, Linux и Windows. В PortAudio есть файлы альтернативных реализаций для поддержки этого интерфейса API для каждой из платформ.

Мне никогда не нужно было копаться в PortAudio для того, чтобы разобраться, что там внутри происходит. Однако, полезно знать, как происходит взаимодействие с PortAudio. Audacity принимает пакеты данных от PortAudio (запись) и посыпает их в PortAudio (воспроизведение). Стоит взглянуть на то, как именно происходит отправка и получение пакетов и как это согласуется с чтением и записью на диск и с обновлением экрана.

В одно и то же время происходят несколько различных процессов. Некоторые происходят часто, передавая небольшие объемы данных, и реакция на них должна быть быстрой. Другие происходят реже, передавая большие объемы данных, и точное время, когда это происходит, менее критично. В результате между процессами возникает рассогласование и для его выравнивания используются буферы. Вторую часть картины составляет то, что мы имеем дело с аудио устройствами, жесткими дисками и экраном. Мы не идем глубже и должны работать с тем интерфейсом API, который нам

предоставлен. Хотя для нас было бы лучше, чтобы каждый из наших процессов выглядел одинакового, например, чтобы каждый из них работал через wxThread, но у нас нет такой возможности (рис. 2.4).

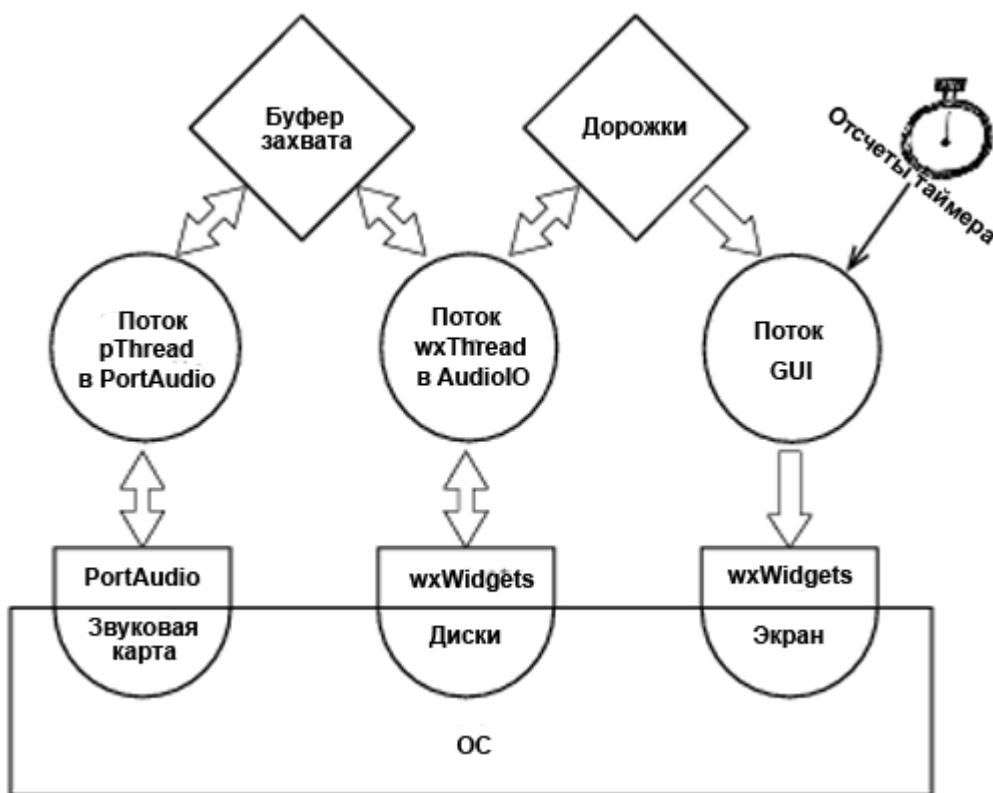


Рис.2.4: Потоки и буферы, используемые при воспроизведении и записи

Один аудио поток запускается кодом PortAudio и непосредственно взаимодействует с аудиоустройством. Это тот поток, который управляет записью и воспроизведением. Этот поток должен реагировать быстро, иначе пакеты будут потеряны. Поток, находящийся под управлением кода PortAudio, называется `audacityAudioCallback`, который, когда происходит запись, добавляет вновь поступившие небольшие пакеты к большему (в пять секунд) буферу захвата. При воспроизведении он берет небольшие кусочки данных из буфера воспроизведения, размер которого равен пяти секундам. Библиотека PortAudio ничего не знает о wxWidgets и поэтому этот поток, созданный PortAudio, является потоком pthread.

Второй поток запускается код в классе AudioIO в Audacity. При записи, AudioIO берет данные из буфера захвата и добавляет их в дорожки Audacity, которые, в конечном счете, будут отображаться на экране. Кроме того, когда будет добавлено достаточно количество данных, AudioIO записывает данные на диск. Этот же поток при воспроизведении аудиозаписей также выполняет операции чтения с диска. Здесь функция `AudioIO::FillBuffers` является ключевой функцией и, в зависимости от настроек некоторых логических переменных, обрабатывает как запись, так и воспроизведение. Важно, чтобы обоих направлениях работала одна функция. Когда происходит «программное воспроизведение», при котором вы накладываете одну запись на другую, которая была записана ранее, одновременно используются части - часть записи и часть воспроизведения. Поток AudioIO мы полностью отдали на откуп операциям ввода/вывода на диск, которые выполняются операционной системой. Во время чтения или записи на диск мы можем останавливаться на неопределенный промежуток времени. Мы не могли вставить эти операции чтения или записи в функцию `audacityAudioCallback`, поскольку она должна реагировать достаточно быстро.

Связь между этими двумя потоками происходит через общие переменные. Поскольку мы контролируем, какие потоки осуществляют записи в эти переменные, нам не требуются более дорогостоящие механизмы типа mutex (механизм управления взаимодействующими процессами).

Как в случае воспроизведения, так и в случае записи, имеется дополнительное требование: Audacity также должна обновлять графический пользовательский интерфейс. Это наименее критичная по времени операция. Обновление происходит в основном потоке графического пользовательского интерфейса и выполняется по таймеру, который тикает двадцать раз в секунду. Этот тик таймера вызывает `TrackPanel::OnTimer`, и, если обнаружено, что необходимо обновление графического интерфейса, то эти обновления выполняются. Такой поток, обслуживающий графический пользовательский интерфейс, создается в `wxWidgets`, а не нашем собственном коде. Особенность его в том, что другие потоки не могут напрямую обновлять графический пользовательский интерфейс. Использование таймера для доступа к графическому пользовательскому интерфейсу для проверки того, нужно ли обновление экрана, позволяет сократить количество перерисовок до уровня, который приемлем для быстро работающих дисплеев и не требует от процессора слишком больших затрат на выдачу изображения.

Хорошим ли проектным решением является использование потока аудиоустройства, потока буфера/диска и потока графического пользовательского интерфейса с таймером, по которому происходят все эти перенаправления аудиоданных? Это специальное решение, представляющее собой три различных потока, которые не заданы в одном абстрактном базовом классе. Впрочем, такая специфика в значительной степени продиктована используемым нами библиотеками. Предполагается, что `PortAudio` создает собственный поток. Во фреймворке `wxWidgets` автоматически создается поток графического пользовательского интерфейса. Наша потребность в использовании потока заполнения буфера обусловлена тем, что нам нужно скорректировать несогласованность между достаточно частыми небольшими пакетами потока аудиоустройства и менее частыми, но большими пакетами дискового устройства. То, что мы используем эти библиотеки, дает нам определенные преимущества. Плата за использование библиотек состоит в том, что мы, в конечном итоге, пользуемся только теми абстракциями, которые они нам предлагают. В результате мы копируем данные из одного места в памяти в другое в гораздо большем объеме, чем это необходимо. В быстрых коммутаторах данных, с которыми я работал, я видел чрезвычайно эффективный код для обработки рассогласования подобного рода, в которых использовались прерывания и, вообще, не использовались потоки. Повсюду передавались указатели на буферы, а не копировались данные. Вы можете применять этот подход только в том случае, если библиотеки, которыми вы пользуетесь, разработаны так, что позволяют работать с абстракцией буфера, обладающей большими возможностями. Воспользовавшись существующими интерфейсами, мы вынуждены пользоваться потоками и вынуждены копировать данные.

2.6. Блочные файлы BlockFile

Одна из проблем, с которыми сталкивается Audacity, это поддержка операций вставки и удаления в аудиозаписях, длина которых может составлять часы. Записи могут легко стать настолько длинными, что не будут помещаться в доступную оперативную память. Если аудиозапись находится в одном файле на диске, то вставка аудиофрагмента где-нибудь ближе к его началу может означать, что для того, чтобы освободить место для вставляемого фрагмента, потребуется перемещение большого количества данных. На копирование этих данных на диск потребуется много времени, что означает, что из-за этого Audacity не сможет быстро реагировать на простые изменения.

Решение этой проблемы в Audacity состоит в разделении аудиофайлов на большое количество блочных файлов `BlockFile`, размер каждого из которых может быть равным около 1 МБ. Это основная причина, почему в Audacity есть свой собственный формат звуковых файлов с мастер-файлом с расширением `.aup`. Это файл XML, с помощью которого координируется использование различные блоки. Изменения вблизи к началу длинной аудиозаписи повлияют только на один блок и на мастер-файл `.aup`.

В блочных файлах `BlockFile` соблюден баланс решения двух противоречащих друг-другу задач. Мы можем вставлять и удалять аудиозаписи без чрезмерного копирования, и во время воспроизведения мы при каждом запросе к диску гарантированно получаем достаточно большие куски ау-

диозаписей. Чем меньше размер блоков, тем потенциально больше требуется запросов к диску для выборки одного и того же количества звуковых данных; чем больше блоки, тем больше объем копирования при вставках и удалениях.

Внутри блочных файлов Audacity никогда нет неиспользуемого внутреннего пространства и они никогда не выходят за пределы максимального размера блока. Чтобы так было всегда, мы, когда вставляем или удаляем фрагменты записи, выполняем копирование по частям, равным размеру одного блока. Когда нам не нужен какой-нибудь блочный файл, то мы его удаляем. Ссылки на блочные файлы сохраняются, так что когда мы удаляем некоторый аудиофрагмент, информация об этом фрагменте будет присутствовать в блочных файлах, тем самым позволяя выполнять операции undo до тех, пока мы не сохраним результат. В блочных файлах Audacity никогда не нужен сборщик мусора для освобождения пространства, который нам бы потребовалось использовать в случае, если бы использовали подход «все в одном файле».

Объединение и разбиение больших кусков данных является основными действиями системы управления данными, начиная от B-деревьев в таблицах BigTable компании Google и заканчивая управлением развернутыми связанными списками. На рис.2.5 показано, что происходит в Audacity при удалении фрагмента аудиозаписи, который находится ближе к началу записи.

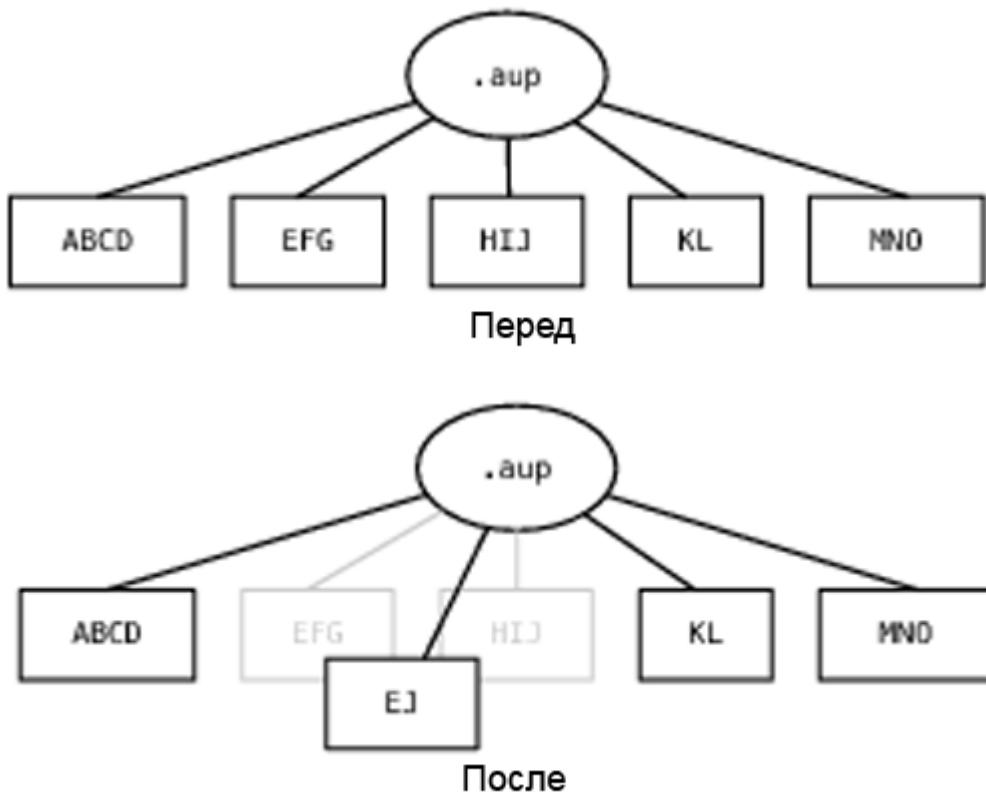


Рис.2.5: Перед удалением в файле .aup и в блок-файлах BlockFile хранится последовательность ABCDEFGHIJKLMNOP. После удаления фрагмента FGHI два блок-файла BlockFile будут объединены в один

Блочные файлы используются не только для аудиозаписей. Есть также блочные файлы, в которых кэшируется обобщающая информация. Если в Audacity делается запрос отображать на экране запись длинною в четыре часа, то невозможно повторно обрабатывать всю аудиозапись каждый раз, когда происходит перерисовка экрана. Вместо этого используется обобщающая информация, в которой приведены максимальная и минимальная амплитуды аудиосигнала для всех фрагментов записи. Когда масштаб увеличивается, Audacity делает перерисовку с использованием фактических фрагментов записей. Когда масштаб уменьшается, Audacity делает перерисовку с использованием обобщающей информации.

Отличительная особенность системы блочных файлов BlockFile состоит в том, что блоки не обязательно должны быть файлами, созданными в Audacity. Они могут быть ссылками на подразделы аудио файлов, таких как временные промежутки аудиозаписей, хранящихся в формате .wav. Пользователь может создать проект Audacity, импортировать аудиозаписи из файла .wav и смикшировать их с несколькими, создав при этом только блочные файлы с обобщающей информацией. При этом сохраняется дисковое пространство и экономится время копирования аудиозаписей. Однако, все говорят, что это довольно плохое решение. Слишком многие из наших пользователей удаляют исходный аудиофайл .wav, думая, что полная его копия будет находиться в папке проекта Audacity. Это не так, поэтому без оригинального файла .wav аудио проект больше воспроизвести не удастся. В настоящее время Audacity по умолчанию всегда копирует импортированную аудиозапись, создавая при этом новые блочные файлы BlockFile.

Решение с использованием блочных файлов вылилось в проблемы в системах Windows, где из-за большого количества блочных файлов сильно падала производительность. Это было связано с тем, что Windows, как оказалось, намного медленнее обрабатывала файлы, когда их много в одном и том же каталоге, что аналогично проблеме с замедлением работы при большом количестве виджетов. В более поздних вариантах было сделано так, что использовалась иерархия подкаталогов и в каждом каталоге никогда не было более сотни файлов.

Основная проблема со структурой блочных файлов связана с тем, что она открыта для конечных пользователей. Мы часто слышим от пользователей, что они переместили файл .aup и не поняли, что также должны были переместить папку, содержащую все блочные файлы. Было бы лучше, если проект Audacity представлял собой один файл, а Audacity взяла на себя ответственность за то, как использовать пространство внутри файла. Во всяком случае это привели бы к увеличению, а не к уменьшению производительности. Основное, что для этого нужно, это - сборщик мусора. Более простой подход состоит в копировании блоков в новый файл при его сохранении в том случае, если в файле не используется более чем некоторый заранее определенный процент памяти.

2.7. Использование сценариев

В Audacity есть экспериментальный плагин, поддерживающий несколько языков сценариев. Он предоставляет интерфейс для сценариев, реализованный поверх конвейера, имеющего имя. Команды сценариев вводятся в текстовом формате, в том же формате выдаются ответы. С помощью языка сценариев можно писать текст и читать текст из именованного конвейера и, таким образом, управлять Audacity. В конвейер не нужно направлять аудиозаписи и другие большие объемы данных (рис. 2.6).



Рис.2.6: Плагин сценариев позволяет писать сценарии, которые перенаправляются в конвейер, имеющий имя.

Самому плагину ничего не известно о содержании текстового трафика, который он передает. Он отвечает только за его передачу. Интерфейс плагинов (илиrudиментарная точка расширения), используемый в плагине сценариев и предназначенный для подключения к Audacity, просто в текстовом формате передает команды Audacity. Поэтому плагин сценариев небольшой, а основное содержится в коде конвейера.

К сожалению из-за конвейера возникает проблема с безопасностью, аналогичная той, которая есть в соединениях TCP/IP, и мы по соображениям безопасности исключили из Audacity соединения TCP/IP. Чтобы уменьшить этот риск, плагин был сделан в виде дополнительной DLL. Вы должны принять сознательное решение, чтобы получить и использовать его, и он поставляется с соответствующим предупреждением, касающимся безопасности.

После того, как функция сценария уже была запущена, то согласно предложению, появившемуся на странице запросов нашей wiki-документации, нам бы следовало рассмотреть возможность использования стандарта D-Bus, имеющегося в KDE, для того, чтобы предоставить механизм межпроцессных вызовов с использованием протокола TCP/IP. Мы уже начали идти по другому пути, но, возможно, по-прежнему есть смысл адаптировать интерфейс, который мы сделали, к D-Bus.

Истоки возникновения кода сценария

Возможность использования скриптов возникла из попытки одного энтузиаста адаптировать Audacity для решения особой задачи, что вело к выделению в разработке отдельной ветки. Эти особенности, вместе называемые как CleanSpeech, были предназначены для преобразования проповедей в формат mp3. В CleanSpeech были добавлены новые эффекты, например, усечение пауз — в аудиозаписи находятся и вырезаются длительные паузы, а также применяется определенная последовательность эффектов по удалению существующих шумов, нормализации и преобразования в mp3 целой группы аудиозаписей. Для этого нам потребовались специальные функциональные возможности, но то, как они были написаны, было слишком специальным случаем в Audacity. Перенос их в основную часть Audacity потребовало бы от нас закодировать использование меняющихся, а не фиксированных последовательностей. Меняющаяся последовательность позволила бы использовать любой из эффектов, указав его с помощью справочной таблицы имен команд, а также класса Shuttle, который сохранял бы параметры команды в текстовом формате в пользовательских настройках. Такая функция называется цепочкой команд потоковой обработки (*batch chains*). Мы совершенно сознательно воздержались от добавления условий и вычислений с тем, чтобы не изобретать специальный язык сценариев.

В ретроспективе, усилия, затраченные на то, чтобы избежать выделения отдельной ветки, оказались правильными. Режим CleanSpeech все еще есть в Audacity, который можно включить при помощи изменения пользовательских настроек. При этом также упрощается пользовательский интерфейс и убираются расширенные возможности. Упрощенная версия Audacity потребовалась для других целей, в первую очередь, в школах. Проблема в том, что каждый по-своему считает, какие возможности являются расширенными, а какие — незаменимыми. Мы впоследствии реализовали простую хитрость, представляющую собой специальный вариант отображения. Когда выполняется отображение пункта меню, который начинается с "#", то этот пункт не показывается в меню. Таким образом, те, кто хотят уменьшить меню, могут сделать этот выбор самостоятельно и без перекомпиляции — более общим и менее инвазивным способом, чем использование в Audacity флага mCleanspeech, который мы со временем сможем полностью удалить.

Работа с CleanSpeech дала нам цепочки команд потоковой обработки и возможность сокращать паузы. Обе возможности внесли дополнительные улучшения. Цепочки команд потоковой обработки непосредственно привели к возможности использования сценариев. Что, в свою очередь, запустило процесс адаптации в Audacity поддержки плагинов общего назначения.

2.8. Эффекты режима реального времени

В Audacity нет эффектов режима реального времени, то есть звуковых эффектов, которые по требованию рассчитываются тогда, когда аудиозапись воспроизводится. Вместо этого в Audacity применяется эффект и нужно ждать его завершения. Одними из наиболее частых требований к Audacity остается добавление эффектов режима реального времени и воспроизведение аудиоэффектов, которые должны происходить в фоновом режиме, а реакция пользовательского интерфейса должна оставаться быстрой.

Проблема, с которой мы имеем дело, состоит в том, что то, что может быть эффектом реального времени на одной машине, может не работать как режим реального времени на другой более медленной машине. Audacity работает на разнообразных машинах. Нам больше подходит, если функциональные возможности снижаются постепенно. На медленных машинах нам бы все еще хотелось иметь возможность запросить эффект, который был бы применен ко всей дорожке, а затем прослушать обработанную аудиозапись где-нибудь ближе к середине дорожки после некоторого ожидания, которое потребуется Audacity с тем, чтобы разбраться, какая часть должна проигрываться первой. На машине, которая слишком медленная для воспроизведения эффекта в режиме реального времени, нам бы нужно было прослушивать аудиозапись до тех пор, пока мы не доберемся до фрагмента, который отображен на экране. Чтобы это сделать, нам бы пришлось разрешить, чтобы аудиоэффекты тормозили пользовательский интерфейс и чтобы порядок обработки аудио блоков был бы только строго слева направо.

Относительно недавнее дополнение в Audacity, которое называется *загрузкой по требованию* (*on demand loading*), хотя оно и не связано с аудиоэффектами, обладает многими элементами, которые нам требуются для эффектов режима реального времени. При импорте аудиофайла в Audacity, теперь можно в фоновом режиме создавать блочные файлы с обобщающей информацией. Пока аудиозапись продолжает загружаться, Audacity будет отображать те места аудиозаписи, которые еще не обработаны, как закрашенные серо-голубыми диагональными полосками, и будет продолжать отвечать на множество пользовательских команд. Блоки не обязательно должны обрабатываться последовательно слева направо. Замысел в том, этот же самый код может пригодиться для эффектов режима реального времени.

Загрузка по требованию предлагает нам эволюционный подход к добавлению эффектов режима реального времени. Это шаг, который позволяет избежать некоторых сложностей при создании самих эффектов режима реального времени. Для эффектов режима реального времени также дополнительно потребуется взаимное наложение между блоками, поскольку в противном случае такие эффекты, как эхо, не будут правильно подключаться. Мы также должны позволить менять параметры во время проигрывания аудиозаписи. Благодаря тому, что сначала сделана загрузка по требованию, код стал использоваться на более раннем этапе, чем это могло бы быть. Благодаря фактическому использованию кода возникнет обратная связь и появятся уточнения.

2.9. Заключение

В предыдущих разделах данной главы рассказано, как хорошая структура способствует развитию программы или как отсутствие хорошей структуры этому мешает.

- Интерфейсы API других разработчиков, такие как PortAudio и wxWidgets, принесли нам огромную пользу. Они предложили нам код, на базе которого была сделана работа, и позволили абстрагироваться от различий, имеющихся во многих платформах. Единственное, чем мы за это заплатили, это то, что у нас слишком мало возможностей выбирать абстракции. У нас не совсем красивый код, используемый для воспроизведения и записи, т.к. мы должны были обрабатывать потоки тремя различными способами. В коде также гораздо больший объем копирования данных, чем это могло бы быть в случае, если бы мы управляли абстракциями.

- Интерфейс API, предложенный нам в wxWidgets, соблазнил нас написать несколько громоздкий код, которому следовать трудно. Наше решение состояло в добавление фасада перед wxWidgets, что предоставило нам абстракции, которые нам были нужны, и дало более чистый код.
- В панели TrackPanel в Audacity нам потребовалось выйти за пределы возможностей, которые можно было бы легко получить в существующих виджетах. В результате мы создали нашу собственную специальную систему. Есть средства освобождения панели с виджетами, с механизмом изменения размеров и с логически различными объектами уровня приложения, использование которых выходит за рамки панели TrackPanel.
- Структурные решения являются более широкими, чем решения, касающиеся того, как структурировать новые возможности. Столь же важным может оказаться решение о том, что не должно быть в программе. Оно может привести к чистому и безопасному коду. Приятно получить преимущества скриптовых языков, таких как Perl, и не выполнять работу по поддержанию нсвоей собственной копии языка. Структурные решения также принимаются с учетом планов будущего развития. Предполагается, что наши модульная система, находящаяся в зачаточном состоянии, позволит нам больше экспериментировать, т. к. эксперименты стали безопаснее. Предполагается, что загрузка по требованию будет эволюционным шагом на пути к обработке по требованию эффектов режима реального времени.

Чем больше смотришь, тем очевиднее, что Audacity является результатом коллективной работы. Сообщество — это больше, чем просто все те, кто непосредственно сотрудничает друг с другом, поскольку зависят от библиотек, у каждой из которых есть свое собственное сообщество с собственными экспертами в своих вопросах. Читая о структурной смеси в Audacity, вероятно, не следует удивляться, что когда сообщество развивается, в него привлекаются новые разработчики, причем с совершенно разными уровнями квалификации.

У меня нет никаких сомнений в том, что то, какое сообщество стоит за Audacity, отражено в сильных и слабых сторонах кода. Более закрытые группы могли бы написать более качественный код, чем тот, который есть у нас, и более последовательно, но при меньшем количестве участников было бы гораздо труднее реализовать весь спектр возможностей Audacity.

Примечания

1. Единственным исключением из этого является язык Найквиста на базе Lisp (*Lisp-based Nyquist language*), который был встроен в Audacity с самого начала. Мы хотели бы сделать его отдельным модулем, поставляемым в комплекте с Audacity, но нам не хватило времени сделать эти изменения.

3. Командная оболочка Bourne-Again Shell

3.1. Введение

Командная оболочка Unix предоставляет интерфейс, который позволяет пользователю при помощи запуска команд взаимодействовать с операционной системой. Но в командной оболочке также представлен довольно богатый язык программирования: есть конструкции управления порядком выполнения команд, изменения этого порядка, организации циклов, условного исполнения; есть также основные математические операции, именованные функции, строковые переменные, а также двунаправленная связь между командной оболочкой и командами, которые в ней вызываются.

Командные оболочки можно использовать в интерактивном режиме, обращаясь к ним из терминала или эмулятора терминала, например, из xterm, или неинтерактивно, читая команды из файла. В большинстве современных командных оболочек, в том числе и в bash, есть средства редактирования командной строки, в которых с помощью команд, похожих на команды emacs или vi, можно

манипулировать с командной строкой, пока она еще не введена, а также есть различные варианты сохранения команд в списке уже выполненных команд.

Обработка данных в bash очень похожа на работу конвейера командной оболочки: после того, как данныечитываются из терминала или из скрипта, они проходят через ряд стадий, трансформируясь на каждой из них, до тех пор, пока командная оболочка наконец не выполнит команду и не получит код состояния, возвращаемый командой.

В данной главе мы рассмотрим основные компоненты bash - обработку входных данных, синтаксический анализ, различные раскрытия слов и прочую обработку команд, а также исполнение команд, если их рассматривать с точки зрения конвейерной обработки. Эти компоненты выступают в роли конвейера, который считывает данные с клавиатуры или из файла, превращая их в выполненные команды.

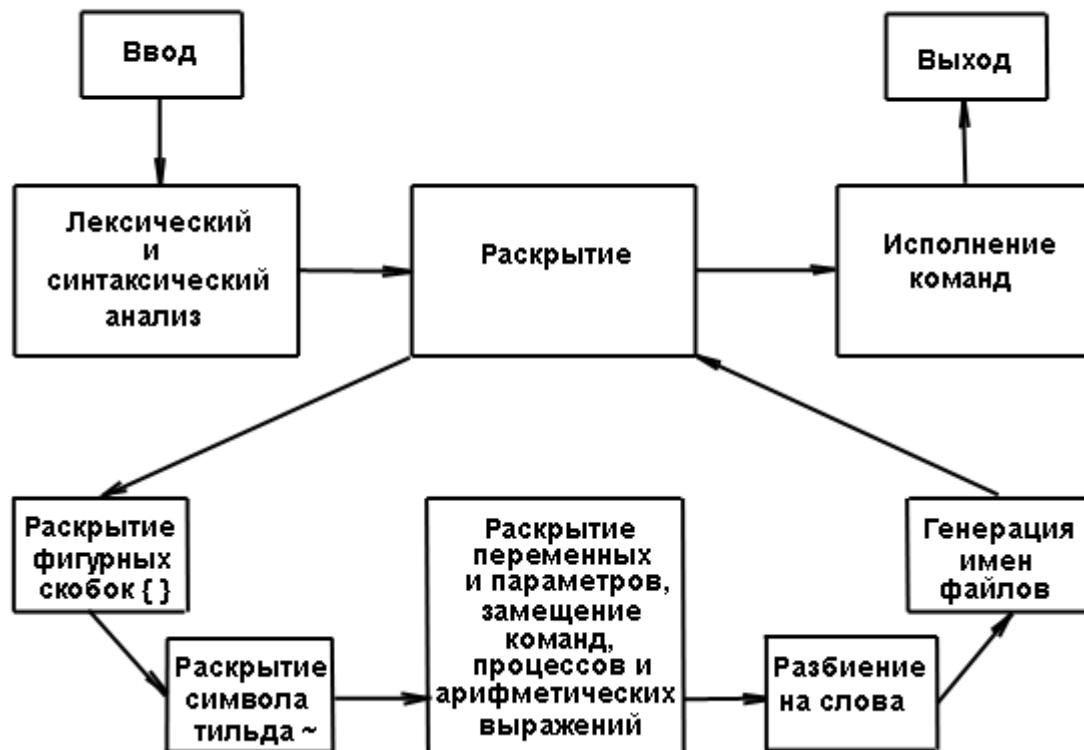


Рис. 3.1: Компонентная архитектура bash

3.1.1. Bash

Bash является командной оболочкой, которая появилась в операционной системе GNU, обычно реализуемой поверх ядра Linux, а также в некоторых других операционных системах и, в первую очередь, в Mac OS X. По сравнению с ушедшими в историю версиями оболочки sh, в bash улучшены функциональные возможности, касающиеся как интерактивного режима, так и возможностей программирования.

Название является сокращением от Bourne-Again SHell, каламбура объединения имени Стивена Борна (Stephen Bourne - автор прямого предшественника используемой в текущих версиях Unix командной оболочки /bin/sh, который появился в версии Bell Labs Seventh Edition Research системы Unix) с понятием перерождения через новую реализацию. Подлинным автором командной оболочки bash был Брайан Фокс (Brian Fox), сотрудник фонда Free Software Foundation. Я работаю

в качестве волонтера в университете Case Western Reserve University в Кливленде, штат Огайо, и в настоящее время являюсь разработчиком bash, а также осуществляю его поддержку.

Точно также как и другое программное обеспечение GNU, bash является полностью переносимым. В настоящее время он работает практически на любой версии Unix, а также на нескольких других операционных системах — есть независимые порты bash, например, для сред Cygwin и MinGW, поддерживаемых в Windows, также порты bash входят в состав дистрибутивов для Unix-подобных операционных систем, например, QNX и Minix. Чтобы собрать и запустить bash, необходима только среда Posix, например, такая, что предоставлена фирмой Microsoft в составе сервисов Services for Unix (SFU).

3.2. Синтаксические единицы и примитивы

3.2.1. Примитивы

Что касается bash, то в ней есть три основных вида лексем: зарезервированные слова, слова и операторы. Зарезервированными словами являются такие, которые в командной оболочке и в ее языке программирования имеют особое значение; как правило, эти слова применяются для написания конструкций, определяющих последовательность выполнения действий, например, `if` и `while`. Операторы состоят из одного или нескольких метасимволов - символов, которые, сами по себе, имеют в командной оболочке особое значение, например, `|` и `>`. Остальные данные, вводимые в командной оболочке, представляют собой обычные слова, некоторые из которых в зависимости от того, где в командной строке они находятся, имеют особое значение — например, инструкции присваивания или числа.

3.2.2. Переменные и параметры

Как и в любом языке программирования, в командной оболочке есть переменные: это имена, на которые ссылаются при хранении данных и при выполнении над ними операций. В командной оболочке можно пользоваться обычными переменными, которые могут задавать пользователи, и некоторыми встроенными переменными, которые называются параметрами. В командной оболочке параметры, как правило, отражают некоторые аспекты внутреннего состояния командной оболочки, и устанавливаются автоматически, или в результате побочного эффекта при выполнении другой операции.

Значениями переменных являются строки. Некоторые значения, в зависимости от контекста, трактуются специальным образом; об этом будет рассказано позже. Переменные назначаются с помощью инструкций вида `name=value`. Значение `value` является необязательным; если оно не указано, то переменной по имени `name` присваивается пустая строка. Если значение указано, оболочка раскрывает значение и присваивает его переменной по имени `name`. В зависимости от того, задано ли значение переменной или нет, оболочка может выполнять разные операции, однако единственным способом задать переменной значение является присваивание. Переменные даже в случае, если они были объявлены и для них были заданы атрибуты, но которым значение еще не присвоено, считаются неопределенными (имеют значение `unset`).

Слово, начинающееся с символа доллара, является переменной или ссылкой на параметр. Слово, в том числе и символ доллара, заменяется на значение переменной с этим именем. В командной оболочке имеется богатый набор операторов, предназначенный для раскрытия значений, начиная от простой замены значения и до изменения или удаления частей значения переменной, соответствующих некоторому образцу.

Есть положения о локальных и глобальных переменных. По умолчанию, все переменные являются глобальными. Любая простая команда (наиболее знакомый тип команды - название команды и необязательные набор аргументов и ссылок) может предваряться набором операторов присваивания,

в этом случае переменные существуют только для этой команды. В командной оболочке есть реализации хранимых процедур или функций командной оболочки, в которых могут быть локальные переменные этих функций.

Типизация переменных минимальна: в дополнение к обычным строковым переменным есть целые числа (тип `integer`) и массивы (тип `integer`). Переменные целочисленного типа трактуются как числа: любая присваиваемая им строка раскрывается как арифметическое выражение и полученный результат присваивается переменной в качестве значения. Массивы могут быть индексные или ассоциативные; в индексных массивах в качестве индексов используются числа, а в ассоциативных массивах - произвольные строки. Элементы массива являются строками, которые, если это необходимо, можно рассматривать как целые числа. Элементами массива не могут быть другие массивы.

В `bash` для запоминания переменных и доступа к ним используются хэш-таблицы, а для реализации областей видимости переменных - связные списки, состоящие из таких хэш-таблиц. Есть различные области видимости переменных для вызовов функций командной оболочки и временные области видимости для переменных, значения которых устанавливается в инструкциях присваивания, предшествующих команде. Когда, например, эти инструкции присваивания предшествуют команде, встроенной в командную оболочку, оболочка должна следить за тем порядком, в котором следует обращаться по ссылкам на эти переменные, и связные области видимости позволяют `bash` это делать. В зависимости от того, какой уровень вложенности используется, может потребоваться просмотреть на удивление много областей видимости.

3.2.3. Язык программирования командной оболочки

Простая команда командной оболочки, которая более всего известна большинству читателей, состоит из имени команды, например, `echo` или `cd`, и списка, состоящего из нуля или большего числа аргументов и перенаправлений. Перенаправления позволяют пользователю командной оболочки управлять в вызываемых командах вводом и выводом данных. Как отмечалось выше, пользователи могут для простых команд определять локальные переменные.

С помощью зарезервированных слов вводятся более сложные команды командной оболочки. Есть конструкции, обычные для любого высокоуровневого языка программирования, например, `if-then-else`, `while`, цикл `for` для итерации по списку значений, а также арифметический цикл `for`, похожий на используемый в языке С. С помощью этих более сложных команд командная оболочка может выполнить команду или другую проверку условия и, в зависимости от полученного результата, выполнять различные операции, либо может повторять выполнение команды много раз.

Одним из подарков, который Unix принес в компьютерный мир, является конвейер: линейный список команд, в котором выход одной команды в списке становится входом следующей команды. В конвейере можно использовать любую конструкцию командной оболочки; не редкость видеть конвейеры, в которых в цикле обрабатываются данные, выдаваемые командами.

В `bash` реализован механизм, который позволяет, когда вызывается команда, перенаправлять стандартный входной поток, стандартный выходной поток и стандартный поток ошибок в другой файл или процесс. Программисты, пользующиеся командной оболочкой, также могут в среде текущей оболочки осуществлять перенаправление в открытые и закрытые файлы.

`Bash` позволяет запоминать программы, работающие в командной оболочке, и использовать их более одного раза. Есть два способа именования групп команд - создать функцию командной оболочки или создать скрипт командной оболочки — и выполнить их точно также, как и любую другую команду. Функции командной оболочки объявляются с использованием специальных синтаксических правил, а затем запоминаются и выполняются в контексте той же самой командной оболочки; скрипт командной оболочки создается при помощи записи команд в файл и выполнение этого файла представляет собой запуск нового экземпляра командной оболочки, которая интер-

претириует этот файл. Функции обычно выполняются в командной оболочке в том же самом контексте исполнения, из которой они были вызваны; однако скрипты, поскольку они интерпретируются в новом экземпляре командной оболочки, могут совместно использовать только те данные, которые в среде командных оболочек могут передаваться между процессами.

3.2.4. Дополнительное замечание

Когда вы будете читать дальше, имейте в виду, что все возможности командной оболочки реализованы с использованием всего лишь нескольких структур данных: массивов, деревьев, односвязных и двусвязных списков и хэш-таблиц. Почти все конструкции, имеющиеся в командной оболочке, реализованы с использованием этих примитивов.

Основной структурой данных, используемой командной оболочкой для передачи информации от одного этапа к другому, а также для работы с элементами данных на каждом этапе обработки, является структура WORD_DESC:

```
typedef struct word_desc {
    char *word;           /* Zero terminated string. */
    int flags;            /* Flags associated with this word. */
} WORD_DESC;
```

Слова, например, объединяются в списки аргументов с помощью простых связанных списков:

```
typedef struct word_list {
    struct word_list *next;
    WORD_DESC *word;
} WORD_LIST;
```

Списки слов вида WORD_LIST используются в командной оболочке повсюду. Простая команда является списком слов, результат раскрытия выражения является списком слов, а в каждой встроенной команде список слов используется в качестве аргументов.

3.3. Обработка входных данных

Первой стадией конвейерной обработки в bash является обработка входных данных: из терминала или из файла берутся символы, из них формируются строки, а затем строки передаются в синтаксический анализатор командной оболочки для их преобразования в команды. Как вы и ожидали, строки представляют собой последовательности символов, заканчивающиеся символами новой строки.

3.3.1. Readline и редактирование командной строки

Bash, когда он находится в интерактивном режиме, читает входные данные с терминала, либо, в противном случае, из файла-скрипта, указанного в качестве аргумента. В интерактивном режиме bash позволяет пользователю с помощью известных последовательностей нажатий клавиш и команд редактирования, похожих на те, что есть в системе Unix в редакторах emacs и vi, редактировать командные строки, которые он набрал.

В bash для редактирования командных строк используется библиотека readline. В ней есть функции, позволяющие пользователям редактировать строки команд, сохраняющие строки команд по мере их ввода, повторно обращающиеся к ранее набранным командам, а также раскрывающие команды по списку истории команд наподобие того, как это сделано в csh. Bash является основным клиентским приложением readline и они разрабатываются вместе, но в readline нет кода, зависящего от bash. Библиотека readline используется во многих других проектах для реализации интерфейса редактирования строк, вводимых с терминала.

Readline также позволяет пользователям связывать последовательности нажатий клавиш неограниченной длины с любой из команд из большого количества команд, имеющихся в readline. В readline есть команды для перемещения курсора вдоль строки, вставки и удаления текста, поиска предыдущих строк и автозавершения частично набранных слов. Помимо этого, пользователи могут определять макросы, которые являются строками символов, вставляемых в командную строку в ответ на нажатие последовательности клавиш; в макросах используется тот же самый синтаксис, что и при связывании последовательностей клавиш с командами. Макросы предоставляют пользователям библиотеки readline средства простой подстановки строк и сокращения усилий при вводе данных.

Структура readline

С точки зрения структуры readline представляет собой цикл "чтения / диспетчеризации / исполнения / повторного отображения". Библиотека читает символы с клавиатуры с помощью операции `read` или другой эквивалентной, либо получает их из макроса. Каждый символ рассматривается как индекс в таблице раскладки клавиатуры или таблицы диспетчеризации. Хотя в качестве индексов используются одиночные восьмибитовые символы, содержимое каждого элемента таблицы раскладки может использоваться для различных целей. Символы могут использоваться для доступа к дополнительным таблицам клавиатурных раскладок, в которых могут быть заданы многосимвольные последовательности нажатий клавиш. Если выясняется, что символ является некоторой командой readline, например, командой `beginning-of-line` (начало строки), то будет выполнена эта конкретная команда. Символ, связанный с командой `self-insert` (самоподставляемый), запоминается в буфере редактирования. Также можно связать последовательность нажатия клавиш с некоторой командой, причем в качестве этой команды использовать последовательность нескольких объединенных вместе различных команд (возможность, добавленная сравнительно недавно); в таблице раскладки есть специальный индекс, указывающий, что была сделана такая привязка. Привязка последовательности нажатия клавиш к макросам позволяет достичь еще большой гибкости: от вставки в командную строку произвольных строк и до создания горячих клавиш для сложных последовательностей операций редактирования. Библиотека readline запоминает каждый символ, связанный с командой `self-insert` (самоподставляемый), в буфере редактирования, который, когда он отображается, может занимать на экране одну или несколько строк.

В библиотеке readline используются символьные буфера и строки, содержащие только тип данных `char` языка C и, если необходимо, из них создаются многобайтовые символы. Тип данных `wchar_t` внутри библиотеки не используется по причинам, связанным со скоростью работы и способами хранения данных, а также из-за того, что код, с помощью которого выполняется редактирование, был создан раньше, чем поддержка многобайтовых символов получила широкое распространение. Когда в локали поддерживается использование многобайтовых символов, то readline автоматически считывает целиком весь многобайтовый символ и помещает его в буфер редактирования. Можно связать многобайтовые символы с командами редактирования, но нужно связывать такой символ как последовательность нажатия клавиш; это возможно, но сложно и, как правило, не требуется. Например, в существующих наборах команд emacs и vi многобайтовые символы не используются.

Как только в команде редактирования будет окончательно определена последовательность нажатий клавиш, readline обновит изображение, выдаваемое на дисплей, для того, чтобы отобразить эти результаты. Это происходит независимо от того, будут ли результаты работы команды в виде символов вставлены в буфер, измениться ли позиция, в которой выполняется редактирование, или будет ли частично или полностью заменена строка. Некоторые связываемые команды редактирования, например, те, что изменяют файл истории команд, не будут вызывать никаких изменений в содержимом буфере редактирования.

Хотя процесс обновления изображения, выдаваемого на терминал, и кажется простым, он состоит из ряда действий. Библиотека readline должна следить за тремя вещами: за текущим содержимым буфера символов, отображаемых на экране, за обновлениями содержимого этого буфера изобра-

жений и за фактически отображаемыми символами. Когда имеются многобайтовые символы, отображаемые символы не соответствуют точно содержимому буфера и средства обновления изображения должны это учитывать. Когда происходит обновление изображения, readline должна сравнить содержимое буфера текущего изображения с обновленным буфером, выявить различия, и решить, как с учетом обновлений, имеющихся в буфере, наиболее эффективно обновить изображение. Эта проблема была предметом серьезного исследования на протяжении многих лет (проблема корректировки вида "строка в строку"). Подход, используемый в readline, состоит в выявлении начала и конца той части буфера, которая отличается, вычисления затрат на обновление только этой части, в том числе на перемещение курсора назад или вперед (например, потребуется ли больше затрат для того, чтобы выдать на терминал команды, которые удалят символы, а затем вставят новые, вместо простой перезаписи текущего содержимого экрана?), выполнении самого меньшего по затратам варианта обновления, а затем, если это необходимо, очистки — удаления всех символов, оставшихся в конце строки, и установке курсора в нужном месте.

Механизм обновления изображения является, без сомнения, одной из частей readline, которые подверглись наибольшим переделкам. Много изменений было сделано для расширения функциональных возможностей — наиболее значимой является возможность использовать в запросе на ввод команды неотображаемые символы (которые, например, изменяют цвет символов), а также обрабатывать символы, занимающие более одного байта.

Readline возвращает содержимое буфера редактирования в приложение, из которого оно было вызвано и которое затем должно сохранить результаты, возможно измененные, в списке истории команд.

Расширение функциональных возможностей readline в приложениях

Точно также как пользователям в readline предлагаются различные способы настройки и расширения функциональных возможностей, используемых по умолчанию, так и приложениям предоставляется ряд механизмов, позволяющих расширить набор возможностей, предлагаемых по умолчанию. Во-первых, подключаемые функции readline получают доступ к стандартному набору аргументов и возвращают определенный набор результатов, что позволяет приложениям легко расширять возможности readline своими собственными функциями. Например, в bash добавлено более тридцати команд, используемых при связывании: от автозавершения конкретных слов и до интерфейсов к командам, встроенным в командную оболочку.

Во втором случае readline позволяет изменить свое поведение за счет использования повсеместно имеющихся указателей на функции, осуществляющие перехват управления (hook function), имена и интерфейс вызова которых хорошо известны. Приложениям разрешено подменять некоторые внутренние фрагменты кода readline, вставлять перед работой readline некоторые собственные функции и выполнять преобразования, необходимые конкретному приложению.

3.3.2. Обработка входных данных в неинтерактивном режиме

Когда командная оболочка не пользуется библиотекой readline, она для получения входных данных будет использовать либо `stdio`, либо свои собственные подпрограммы буферированного ввода. Если командная оболочка находится в неинтерактивном режиме, то использование пакета буферированного ввода, который есть в `bash`, более предпочтительно, чем `stdio`, из-за нескольких своеобразных ограничений, которые связаны в Posix с тем, что следует делать при вводе данных: на вход командной оболочки должны поступать только данные, необходимые для анализа команд, а все остальное должно передаваться исполняемым программам. В частности это важно, когда оболочка считывает скрипт из стандартного входного потока. Командная оболочка может буферировать входные данные в том объеме, сколько это будет необходимо, и так долго, пока в файле сразу после того, как будет проанализирован последний символ, не будет выполнен откат обратно. С практической точки зрения это значит, что когда данныечитываются из устройств, в которых нет возможности выполнять поиск, например, из конвейеров, командная оболочка должна считывать

скрипта символ за символом, но когда чтение происходит из файла, в буфер можно записывать столько символов, сколько будет необходимо.

Если оставить эти особенности в стороне, то на выходе процесса обработки, выполняемого командной строкой при неинтерактивном вводе входных данных, будет то же самое, что и у readline: буфер символов, заканчивающихся символом новой строки.

3.3.3. Многобайтовые символы

Обработка многобайтовых символов была добавлена в командную оболочку значительно позже первоначальной реализации оболочки, причем это было сделано таким образом, чтобы свести к минимуму влияние этого добавления на уже существующий код. Когда установлена локаль, в которой есть поддержка многобайтовых символов, то командная оболочка записывает свои входные данные в буфер, состоящий из байтов (тип `char` языка C), но трактует эти байты как потенциально возможные многобайтные символы. В readline известно, как выводить многобайтовые символы (здесь важно знать, сколько позиций на экране занимает многобайтный символ и сколько байтов нужно считать из буфера, когда символ отображается на экране), как перемещаться по строке вперед и назад в случае, когда перемещение происходит не на один байт за один раз, и так далее. Во всем другом многобайтовые символы не оказывают большого влияния на процесс ввода данных. Другие части командной оболочки, которые будут описаны ниже, должны знать о том, что используются многобайтовые символы, и учитывать это при обработке входных данных.

3.4. Анализ

Первой частью работы, выполняемой при анализе данных, является лексический анализ: поток символов в соответствие со смыслом слов разделяется на слова. Слово является основной единицей, с которой работает синтаксический анализатор. Слова являются последовательностями символов, разделенных метасимволами, которыми могут быть простые разделители, например, пробелы и символы табуляции, или символы, являющиеся специальными в языке командной оболочки, например, символ точки с запятой и символ амперсанда.

Одна исторически сложившаяся проблема, касающаяся командной оболочки, состоит в том, как сказал Том Дафф (Tom Duff) в своей статье о `rc` - командной оболочке системы Plan 9, что никто не знает, что такое грамматика оболочки Борна. Особой благодарности заслуживает Комитет по стандарту оболочки Posix, который, наконец, опубликовал окончательную редакцию грамматики для оболочки Unix, хотя и в ней есть масса контекстных зависимостей. Эта грамматика не без проблем — в ней запрещены некоторые конструкции, которые были бы без ошибок восприняты давно созданными синтаксическими анализаторами оболочки Борна, но это лучшее, что у нас есть.

Синтаксический анализатор `bash` был создан на основе ранней версии грамматики Posix, и, насколько я знаю, является лишь синтаксическим анализатором командной оболочки в стиле Борна, реализованной с помощью Yacc или Bison. Вследствие этого возник определенный набор трудностей — в действительности грамматика командной оболочки не очень хорошо подходит для синтаксического анализа в стиле yacc и требует более сложного лексического анализа и большего объема взаимодействий между синтаксическим и лексическим анализаторами.

В любом случае, лексический анализатор получает входные строки из readline или другого источника, разбивает их на лексемы, разделяемые метасимволами, идентифицирует лексемы с учетом контекста и передает их в синтаксический анализатор для сборки их в инструкции и команды. Контекст может быть весьма различным — например, слово `for` может быть зарезервированным словом, идентификатором, часть инструкции присваивания, или другим словом, и следующая команда, которая является вполне допустимой:

```
for for in for; do for=for; done; echo $for
```

выдает на терминал слово `for`.

В данный момент настала очередь сделать небольшое отступление, относящееся к использованию алиасов (или синонимов — прим.пер.). Bash позволяет с помощью методики алиасов заменять произвольным текстом первое слово простой команды. Поскольку эта замена полностью лексическая, алиасы можно даже употреблять (или ими злоупотреблять) для того, чтобы изменить грамматику командной оболочки: можно написать алиас, реализующий составную команду, которой нет в bash. Анализатор bash реализует методику алиасов полностью на фазе лексического анализа, тем не менее, синтаксический анализатор должен информировать лексический анализатор, когда раскрытие алиасов не допускается.

Как и во многих других языках программирования, в командной оболочке разрешается перед специальными символами указывать другие специальные символы, отменяющие особенности использования первых специальных символов (т.е. использовать так называемые escape-последовательности символов — прим.пер.), поэтому в командах можно использовать метасимволы, например, `&`. Есть три типа кавычек, каждый из которых немного отличается и позволяет несколько по иному интерпретировать выделенный текст: обратный слеш, который экранирует следующий символ, одинарные кавычки, которые предотвращают интерпретацию всех символов, находящихся внутри кавычек, и двойные кавычки, которые отключают некоторые виды интерпретации, но позволяют выполнять раскрытие некоторых слов (и иначе интерпретировать символы обратного слеша). Лексический анализатор считывает символы и строки, заключенные в кавычки, и не позволяет синтаксическому анализатору искать в них зарезервированные слова или метасимволы. Есть также два особых варианта — `$'...'` и `$"..."`, в которых символы, перед которыми указан обратный слеш, раскрываются точно также, как это делается в строках языка ANSI C, и в которых можно, соответственно, транслировать символы с использованием функций, поддерживающих интернационализацию. Первый вариант используется широко, последний, возможно, из-за того, что для него приведено мало хороших примеров или вариантов его использования, применяется в меньшей степени.

Остальная часть интерфейса между синтаксическим и лексическим анализаторами сравнительно проста. Синтаксический анализатор кодирует определенное количество состояний и использует их совместно с лексическим анализатором для того, чтобы можно было с помощью грамматики выполнить контекстно-зависимый анализ. Например, лексический анализатор классифицирует слова в зависимости от типа лексемы: зарезервированное слово (в соответствующем контексте), слово, инструкция присваивания и так далее. Чтобы это сделать, синтаксический анализатор должен независимо от того, обрабатывается ли многострочный текст (иногда называемых "встроенным документами"), анализируется ли инструкция `case` или команда условного выполнения, обрабатывается ли расширенный шаблон или составная инструкция присваивания, сообщить некоторые сведения о том, насколько далеко вперед продвинулся синтаксический разбор команды.

Большая часть работы, связанной с нахождением конца замещения команды на стадии синтаксического анализа, инкапсулирована в одной функции (`parse_comsub`), которая разбирается со всеми неудобными случаями синтаксиса командной оболочки и в ней гораздо больше кода, читающего лексемы, чем это было бы в оптимальном случае. Эта функция должна знать о встроенных документах, о комментариях командной оболочки, о метасимволах и границах слов, об использовании кавычек и случаях, когда можно использовать зарезервированные слова (так что она знает, когда они должны быть в инструкции `case`); потребовалось время чтобы делать это правильно.

Когда в процессе раскрытия слов необходимо замещение команды, bash для того, чтобы найти правильное окончание конструкции, пользуется синтаксическим анализатором. Это аналогично преобразованию строки в команду с помощью команды `eval`, но в этом случае команда не завершается концом строки. Чтобы выполнить эту работу, синтаксический анализатор должен распознать правую скобку как признак завершения команды; при выводе ряда грамматических правил это приводит к частным случаям и требуется, чтобы лексический анализатор помечал правую скобку (в соответствующем контексте) как символ конца файла EOF. Прежде, чем рекурсивно об-

ращаться к `yyparse`, анализатор также должен уметь сохранять и восстанавливать свое внутреннее состояние, поскольку при замещении команды в процессе ее чтения может потребоваться произвести синтаксический анализ и выполнить часть операции раскрытия. Поскольку в функциях ввода данных реализовано упреждающее чтение, то независимо от того, будет ли `bash` читать данные из строки файла или с терминала с помощью `readline`, следует, наконец, также позаботиться о перемещении указателя входного потока `bash` вправо в нужное место. Это важно не только чтобы не потерять входные данные, но также и для того, чтобы функции, выполняющие замещение команд, создали для исполнения правильную строку.

Аналогичные проблемы встают при программировании автозавершения слов, когда при синтаксическом разборе одной команды может выполнять произвольное количество любых других команд, причем во всех вызовах синтаксического анализатора нужно решать проблему сохранения и восстановления его состояния.

Использование кавычек также является источником несовместимости и обсуждений. Спустя двадцать лет после того, как был опубликован первый стандарт командной оболочки Posix, члены Рабочей группы по стандартам до сих пор обсуждают правильную обработку неясных случаев использования кавычек. Как и прежде, командная оболочка Bourne не поможет ничем, кроме как наблюдать за ней как за эталонной реализацией.

Синтаксический анализатор возвращает единственную структуру на языке C, представляющую собой команду (которая, в случае составных команд, например, циклов, может, в свою очередь, состоять из нескольких других команд), и передает ее на следующий этап работы командной оболочки - этап раскрытия слов. В структуре хранятся объекты, представляющие команду, и списки слов. Большая часть списков слов будет, в зависимости от контекста, преобразована, что описывается в следующих разделах статьи.

3.5. Раскрытие слов

После синтаксического разбора, но до этапа исполнения, многие из слов, сформированные на стадии синтаксического анализа должны быть подвергнуты одной или нескольким операциям раскрытия, так, например, слово `$OSTYPE` будет заменено строкой `"linux-gnu"`.

3.5.1. Раскрытие параметров и переменных

Пользователи лучше всего известно раскрытие переменных. Переменные командной оболочки вводятся с клавиатуры и, за немногими исключениями, трактуются как строки. Операция раскрытия заменяет эти строки и трансформирует их в новые слова и списки слов.

Есть варианты операций раскрытия, которые действуют на значение самой переменной. Программисты могут ими пользоваться для получения подстрок из значения переменной, определения длины строки, удаления различных частей строки, соответствующих при просмотре с начала строки или с ее конца заданному шаблону, замены части строки, соответствующей заданному шаблону, новой строкой, или изменения набора символов.

В добавок есть варианты раскрытия, которые зависят от состояния переменной: в зависимости от того, задано ли переменной значение или нет, могут выбираться различные варианты раскрытия или присваивания различные значения. Например, `${parameter:-word}` будет раскрыто как `parameter` в случае, если значение переменной задано, и как `word` в случае, если оно не задано или задано значение пустой строки.

3.5.2. И многое другое

Bash выполняет много других вариантов операций раскрытия, для каждой из которых есть свои собственные причудливые правила. Сначала обрабатывается раскрытие скобок, которое превращает:

```
pre{one,two,three}post
```

в:

```
preonepost pretwopost prethreepost
```

Также есть замещение команд, что замечательно сочетается с возможностью командной оболочки запускать команды и манипулировать с переменными. Оболочка запускает команду, сохраняет результат ее работы и использует его в качестве значения, используемого в операции раскрытия.

Одна из проблем, связанная с замещением команд, состоит в том, что в этом случае немедленно запускается команда, работающая изолированно, и ожидается ее завершение: в командной оболочке нет простого способа передать команде входные данные. В bash используется возможность, называющаяся замещением процессов и представляющая собой своего рода комбинацию замещения команды и конвейеров командной оболочки, благодаря чему этот недостаток компенсируется. Точно также, как и в случае замещения команд, в bash запускается команда, но она работает в фоновом режиме и bash не ожидает ее завершения. Главное то, что bash открывает конвейер к команде для чтения или записи и указывает в конвейере имя файла, который станет результатом операции раскрытия.

Далее идет раскрытие символа тильды ~. Первоначально предполагалось, что ~alan будет ссылкой на домашний каталог Алана, но за прошедшие годы этот вариант раскрытия сильно расширился и позволяет ссылаться на большое количество различных каталогов.

Наконец, имеется раскрытие арифметических выражений. В \$((expression)) выражение expression должно вычисляться по тем же правилам, что выражения языка C. Результат вычисления выражения становится результатом раскрытия.

Раскрытие переменных это как раз тот случай, когда наиболее очевидной становится разница между одинарными и двойными кавычками. Одинарные кавычки отключают все раскрытия - символы, заключенные в кавычки, передаются через операцию раскрытия без всяких изменений, тогда как в случае двойных кавычек некоторые раскрытия выполняются, а другие — не выполняются. Выполняется раскрытие слов и команд, раскрытие арифметических выражений и выполняется замещение процессов - двойные кавычки только влияют то, как обрабатывается результат, раскрытие скобок и символа тильды не выполняется.

3.5.3. Разбиение на слова

Результат раскрытия слов разбивается на отдельные слова, причем в качестве разделителей слов используются символы, указанные в переменной командной оболочки IFS. Речь идет о том, как командная оболочка преобразует одно слово в несколько слов. Каждый раз, когда в результате выполнения операции раскрытия обнаруживается один из символов, указанных в переменной \$IFS (смотрите в конце статьи примечание 1) bash разбивает слово на два новых слова. Одинарные и двойные кавычки отключают функцию разбиения на слова.

3.5.4. Подстановка

После того, как результаты будут разбиты на слова, командная оболочка проинтерпретирует каждое слово, полученное в результате предыдущих раскрытий, в качестве потенциального шаблона и попытается сопоставить его с существующим полным именем файла, включающим все пути, ведущий к каталогам.

3.5.5. Реализация

Если базовая архитектура командной оболочки допускает распараллеливание конвейеров, то раскрытие слов будет небольшим конвейером, замкнутым самим на себе. На каждом этапе раскрытия слов берется некоторое слово и после того, как оно, возможно, будет преобразовано, оно передается на следующий этап раскрытия. После того, как все этапы раскрытия слова будут пройдены, будет выполнена команда.

Реализация раскрытия слов в bash строится на основе уже ранее описанных структур данных. Слова, выдаваемые синтаксическим анализатором, раскрываются по одному, в результате чего каждое отдельное слово, имеющееся на входе, будет раскрыто на выходе в виде одного или нескольких слов. Структура данных `WORD_DESC` оказалась достаточно универсальной и в ней может храниться вся информация, необходимая для инкапсуляции результатов раскрытия одного слова. Для кодирования информации, используемой на стадии раскрытия слов, а затем передаваемой с этой стадии на следующую, применяются флаги. Например, синтаксический анализатор использует флаг, говорящий на стадиях раскрытия слов и исполнения команд о том, что конкретное слово является инструкцией присваивания командной оболочки; а в коде, осуществляющим раскрытие слов, флаги используются для запрета разбиения на слова или пометки о присутствии заключенной в кавычки строки, имеющей значение `null`, ("`$x`", где `$x` не определено или имеет значение `null`). Использовать для раскрытия всех слов единую строку символов с какой-то кодировкой для представления дополнительной информации, оказалось бы гораздо сложнее.

Как и в синтаксическом анализаторе, в коде, осуществляющем раскрытие слов, обрабатываются символы, для представления которых требуется более одного байта. Например, длина переменной при раскрытии (`$(#variable)`) подсчитывается в символах, а не в байтах, и код в случае использования многобайтовых символов может правильно идентифицировать завершение операций раскрытия и найти специальные символы, используемые при раскрытии.

3.6. Исполнение команд

Стадия выполнения команд внутреннего конвейера bash является тем местом, где происходит реальное действие. Как правило, набор слов, для которых было выполнение раскрытие декомпозируется на имя команды и набор аргументов и передается в операционную систему в виде файла, который читается и исполняется вместе с остальными словами, передаваемыми в остальных элементах структуры `argv`.

В нашем описании внимание до сих пор умышленно сосредотачивалось на том, что Posix обращается к простым командам — тем, у которых есть имя и набор аргументов. Это наиболее распространенный тип команд, но у bash намного больше возможностей.

На входе на стадию выполнения команд используется структура данных, созданная синтаксическим анализатором и заполненная возможно раскрытыми словами. Вот где в игру вступает настоящий язык программирования bash. В языке программирования, как уже указывалось ранее, используются переменные и раскрытия и также реализованы конструкции, наличие которых следовало бы ожидать в языке высокого уровня: циклы, условные инструкции, чередование, объединение в множества, выбор из множеств, условное выполнение с учетом сопоставления с шаблоном, оценка выражений и несколько конструкций более высокого уровня, характерных для командной оболочки.

3.6.1. Перенаправление

Одна из особенностей роли командной оболочки, когда она используется как интерфейс к операционной системе, это делать перенаправление ввода и вывода в команде, которая вызывается. Синтаксис перенаправления является одной из тех составляющих, которые представили первым

пользователям всю сложность командной оболочки: до недавнего времени требовалось, чтобы пользователи следили за дескрипторами файлов, которыми они пользуются, и в случае, когда дескриптор отличался от стандартных потоков ввода, вывода и ошибок, явно указывали его номер.

Недавнее дополнение к синтаксису перенаправления позволяет пользователям вместо того, чтобы давать пользователю самостоятельно выбирать дескриптор файла, дать указание командной оболочке выбрать подходящий дескриптор файла и присвоить его конкретной переменной. Программистам становится проще отслеживать дескрипторы файлов, но это требует дополнительной обработки: оболочка должна сделать в подходящем месте копии дескрипторов файлов и проверять, что они назначены конкретной переменной. Это еще один пример того, как информация передается из лексического анализатора через синтаксический анализатор на стадию выполнения команды: лексический анализатор определяет, что в слове, в котором есть перенаправление, осуществляется присваивание значения переменной; синтаксический анализатор при помощи соответствующих правил грамматического вывода создает объект, используемый при перенаправлении, у которого есть флаг, указывающий, что требуется аргумент; код, осуществляющий перенаправление, получает этот флаг и обеспечивает, чтобы номер дескриптора файла был назначен правильной переменной.

Самой трудной частью реализации перенаправления является запоминание информации, необходимой для отмены перенаправления. В командной оболочке намеренно стирается различие между командами, выполняемыми из файловой системы, что требует создания нового процесса, и командами, которые оболочка выполняет сама (встроенные команды), но, вне зависимости от того, как команда реализована, эффект перенаправления не должен сохраняться после того, как эта команда будет завершена (смотрите в конце статьи примечание 2). Поэтому командная оболочка должна следить за тем, как отменить эффект каждого перенаправления, в противном случае перенаправление выходного потока во внутренней команде изменит стандартный выходной поток самой командной оболочки. В bash известно, как отменять перенаправление каждого типа: либо с помощью закрытия дескриптора файла, который ранее был выделен, либо с помощью создания дубля дескриптора файла и позже восстановления дескриптора с помощью команды `dup2`. Здесь используются те же самые объекты перенаправления, которые были созданы синтаксическим анализатором, а для обработки используются те же самые функции.

Т.к. многократные перенаправления реализованы в виде простых списков объектов, перенаправления, используемые для их отмены, хранятся в отдельном списке. Этот список обрабатывается, когда выполнение команды завершено, но об этом должна позаботиться командная оболочка, поскольку перенаправления, используемые с функцией командной оболочки или со встроенной функцией `"."`, могут продолжать действовать до тех пор, пока функция не будет завершена. Когда происходит обращение к встроенной функции `exec`, причем она не вызывается как команда, то список отмены перенаправлений будет просто удаляться, поскольку перенаправления, связанные с `exec`, запоминаются и хранятся в среде командной оболочки без их отмены.

Другая сложность связана с самой оболочкой bash. В ранее использовавшихся версиях оболочки Bourne пользователю разрешалось обращаться только к дескрипторам 0 - 9, дескрипторы 10 и выше были зарезервированы для внутреннего использования в самой оболочке. В bash это ограничение ослаблено — пользователь может манипулировать дескриптором с любым номером вплоть до предела, обусловленного ограничением на количество открытых в процессе файлов. Это значит, что bash должен следить за дескрипторами файлов, открытых для его собственных внутренних нужд, в том числе и тех, что были открыты внешними библиотеками, а не только непосредственно самой оболочкой, и иметь возможность при необходимости переместить эти дескрипторы. Для этого требуется учитывать многое, в некоторых эвристиках нужно использовать флаг `close-on-exec`, и в течение всего времени, пока выполняется команда, необходимо поддерживать еще один список перенаправлений, который затем будет обработан или будет просто удален.

3.6.2. Встроенные команды

В bash есть ряд команд, которые являются частью самой оболочки. Эти команды выполняются самой командной оболочкой без создания нового процесса.

Самым распространенным мотивом сделать команду внутренней является возможность поддержки или изменения внутреннего состояния командной оболочки. Хорошим примером является команда `cd`; в одном из классических упражнений на вводных занятиях по Unix объясняется, почему команду `cd` нельзя реализовывать как внешнюю.

Встроенные команды bash используют те же самые внутренние примитивы, что и остальная часть командной оболочки. Каждая встроенная команда реализована с помощью функций языка C, которая в качестве аргументов используется список слов. Это те слова, которые поступают со стадии раскрытия строк; встроенные команды рассматривают их как имена команд и аргументы. По большей части, встроенные команды используют те же самые стандартные правила раскрытия, что и другие команды, но с некоторыми исключениями: встроенные команды bash, в которых в качестве аргументов допускаются инструкции присваивания (например, `declare` и `export`), применяют для аргументов с инструкцией присваивания те же самые правила раскрытия, которые оболочка применяет при присваивании значений переменным. Это единственное место, где для передачи информации от одной стадии внутреннего конвейера командной оболочки к другой используется элемент `flags` из структуры `WORD_DESC`.

3.6.3. Выполнение простых команд

Простые команды — это те, которые встречаются чаще всего. Команда ищется в файловой системе, затем — исполняется, а после ее завершения определяется статус завершения, используемый при доступе ко многим другим возможностям командной оболочки.

Присваивание значения переменной в командной оболочке (т.е. слова вида `var=value`) само является своего рода простой командой. Инструкции присваивания могут либо предшествовать имени команды, либо находиться в отдельной командной строке. Если они предшествуют команде, то переменные передаются в исполняемую команду через среду окружения команды (если они предшествуют встроенной команде или функции командной оболочки, то они сохраняются, за немногими исключениями, лишь пока исполняется встроенная команда или функция). Если за инструкциями присваивания нет имени команды, то эти инструкции изменяют состояние командной оболочки.

Если представлено имя команды, которая не является именем функции командной оболочки или встроенной команды, то bash ищет в файловой системе исполняемый файл с таким именем. Для поиска будет использована значение переменной `PATH`, представляющее собой список каталогов, разделенных символами двоеточия. Поиск команд с именами, в которых есть слеш (или символы других разделителей каталогов) не происходит — они сразу передаются на исполнение.

Если команда найдена с использованием переменной `PATH`, то bash сохраняет имя команды и соответствующий полный путь в хеш-таблице, к которой он обратится в следующий раз перед следующим поиском в переменной `PATH`. Если команда не найдена, то bash выполняет функцию со специальным именем, если он ее найдет, а в качестве аргументов возьмет имя команды, которая должна была быть выполнена, и список ее аргументов. В некоторых дистрибутивах Linux это свойство используется для того, чтобы устанавливать недостающие команды.

Если bash находит файл, который должен быть выполнен, то процесс разветвляется на два процессы, создается новая среда исполнения и выполнение файла происходит в этой новой среде. Новая среда исполнения будет точной копией среды исполнения командной оболочки с незначительными отличиями, связанными с сигналами и открытием или закрытием файлов при перенаправлении.

3.6.4. Управление заданиями

Командная оболочка может выполнять команды в приоритетном режиме, в котором она ждет, пока команда завершится и получает статус выхода из команды, или в фоновом режиме, в котором командная оболочка сразу переходит к чтению следующей команды. Благодаря механизму управления заданиями можно перемещать процессы (команды, которые выполняются) между приоритетным и фоновым режимами, а также приостанавливать и возобновлять их исполнение. Для реализации этого в bash вводится понятие задания (job), которое, по существу является командой, исполняемой одним или несколькими процессами. В конвейере, например, для каждого элемента конвейера используется один процесс. Несколько отдельных процессов можно объединить в группу в виде одного задания. Терминалу будет назначен идентификатор группы процессов, связанных с этим терминалом, так что группа процессов, работающая в приоритетном режиме, эта же группа, идентификатор которой, такой же, как и у терминала.

В командной оболочке для реализации управления заданиями используется несколько простых структур данных. Есть структура для представления дочернего процесса, в которой хранится идентификатор процесса, его состояние и состояние, которое он возвращает при завершении. Конвейер является всего лишь простым связанным списком, состоящим из таких структур. Задание очень похоже: есть список процессов, некоторое состояние задания (задание исполняется, приостановлено, завершено и т.д.), и идентификатор группы процессов задания. Список процессов обычно состоит из одного процесса; с заданием связано более одного процесса только для конвейера. Каждое задание имеет уникальный идентификатор группы процессов, а процесс в задании, идентификатор процесса которого такой же, как идентификатор группы процессов задания, называется лидером группы процессов. Текущий набор заданий хранится в массиве, концептуально очень похожем на то, как это представляет пользователь. Состояние задания и состояния выхода из задания формируются путем агрегирования состояний входящих в него процессов и состояний, возвращаемых этими процессами при их завершении.

Как и со многим другим в командной оболочке, сложной частью, касающейся реализации управления заданиями, является учет. Оболочка должна позаботиться о назначении процессов правильным группам процессов, удостовериться, что синхронизировано создание дочернего процесса и его назначение группе процессов и что надлежащим образом задана группа процессов терминала, поскольку группа процессов терминала определяет задание, работающее в приоритетном режиме (и если его не вернуть его обратно группе процессов оболочки, то оболочка сама не сможет прочитать ввод с терминала). Поскольку учет очень сильно ориентирован на процессы, не так легко реализовать составные команды, например, циклы `while` и `for`, поскольку весь цикл должен останавливаться и запускаться как единое целое, и это должно быть сделано в нескольких командных оболочках.

3.6.5. Составные команды

Составные команды состоят из списка одной или нескольких простых команд и начинаются с ключевого слова, например, `if` или `while`. Именно здесь видна и эффективна вся мощь программирования командной оболочки.

Реализация довольно проста. Синтаксический анализатор строит объекты, соответствующие различным составным командам, и интерпретирует их по мере обхода объектов. Каждая составная команда реализуется с помощью соответствующей функции языка C, которая отвечает за выполнение надлежащих раскрытий, исполнения команд так, как это указано, и изменения порядка исполнения в соответствие со статусом возврата команд. В качестве иллюстрации рассмотрим функцию, с помощью которой реализована команда `for`. Она сначала должна раскрыть список слов, следующих за зарезервированным словом `in`. Затем функция должна выполнить итерацию по раскрытым словам, назначая каждое слово соответствующей переменной, а затем исполнить список команд, указанных в теле команды `for`. Команда `for` не должна изменять порядок исполнения в соответствие со статусом возврата команд, но она должна обращать внимание на встроенные команды `break` и `continue`. После того, как все слова в списке будут использованы, произойдет выход из команды `for`. Видно, что реализация, большей частью, весьма близка к описанию команды.

3.7. Усвоенные уроки

3.7.1. Что, по моему мнению, важно

Я потратил более двадцати лет на работу над bash и, как мне бы хотелось думать, обнаружил кое-что интересное. Самое главное, что я не могу переоценить, то, что очень важно иметь подробные журналы изменений. Хорошо, когда вы можете посмотреть в журнал и вспомнить, почему было сделано конкретное изменение. Еще лучше, если вы можете дополнить это изменение конкретным сообщением об ошибке, тестом, который можно будет повторить, или советом.

Если необходимо тщательное регрессионное тестирование, то это именно то, что я бы порекомендовал делать в проекте с самого начала. В bash есть тысячи тестов, охватывающий практически все его неинтерактивные возможности. Мне надо было создавать тесты для интерактивных функций - в Posix они есть в тестовом фреймворке, но не хотелось иметь дистрибутив, в котором мне бы пришлось принять решение его использовать.

Стандарты важны. Bash выигрывает от того, что реализован в соответствие со стандартом. Важно участвовать в стандартизации программы, которую вы реализуете. Кроме обсуждений возможностей и функций, наличие стандарта позволит использовать стандарт в качестве арбитра. Конечно, стандарт также может оказаться плохим, это зависит от самого стандарта.

Внешние стандарты важны, но также хорошо иметь внутренние стандарты. Мне посчастливилось воспользоваться набором стандартов проекта GNU, в которых предлагаются достаточно хорошие практические советы по разработке и реализации программ.

Еще один важный аспект - хорошая документация. Если вы предполагаете, что программой будут пользоваться другие, важно иметь исчерпывающую и ясно написанную документацию. Если программа будет успешно использоваться, то, в конечном итоге, документации для нее будет много, но важно, чтобы разработчик написал свою собственную версию.

Есть очень много хорошего программного обеспечения. Пользуйтесь всем, чем можете: например, в gnulib есть много удобных библиотечных функций (если только вы сможете извлечь их из фреймворка gnulib). Так делают в системах BSD и Mac OS X. Пикассо как-то по случаю сказал: "У великих художников воруют".

Привлекайте к участию сообщество пользователей, но будьте готовы к периодическим критическим замечаниям, некоторые из которых заставят вас сильно задуматься. Активное сообщество пользователей может быть огромным преимуществом, но одно из последствий состоит в том, что эти люди очень страстные. Не воспринимайте это как личное.

3.7.2. Что я должен был бы сделать по-другому

У bash миллионы пользователей. Я знаю о важности обратной совместимости. В некотором смысле обратная совместимость означает, что вам никогда не придется извиняться. Но мир не так прост. Время от времени я вынужден был делать несовместимые изменения, почти из-за каждого из них от пользователей поступало некоторое количество жалоб, хотя у меня всегда было то, что я считал уважительной причиной, будь то замена плохого решения, которое исправляло неправильную работу, или корректировка несоответствий между различными частями командной оболочки. Я должен был бы раньше ввести что-то вроде формальных уровней совместимости bash.

Разработка bash никогда не была особенно открытой. Я привык к использованию промежуточных релизов (например, bash-4.2) и отдельно разработанных патчей. Для этого есть причины: я подстроился под поставщиков с их сроками выпуска релизов, более длинными, чем выпуск релизов в мире бесплатного программного обеспечения и открытого исходного кода, и у меня в прошлом

были проблемы с бета-версиями, которые становились все более распространенными, чем мне было того хотелось. Хотя, если мне пришлось бы начать все заново, я бы предпочел более частые релизы и пользовался одним из вариантов публичного репозитория.

Но этот список был бы неполным, если не коснуться реализации bash. Есть вопрос, к которому я обращался несколько раз, но не приступил к его решению, - это полное переписывание синтаксического анализатора bash с использованием прямого рекурсивного спуска, а не с использованием *bison*. Когда-то я думал, что я должен сделать это с тем, чтобы привести подстановку команд в соответствие с POSIX, но я мог бы сделать это, если бы не изменения, которых было очень много. Если бы я начал писать bash с нуля, я бы, наверное, написал синтаксический анализатор вручную. Это, конечно, сделало бы его в некоторых местах проще.

3.8. Заключение

Bash является хорошим примером большого и сложного куска свободного программного обеспечения. Его преимущество в том, что он разрабатывался более двадцати лет и стал сформировавшимся и мощным проектом. Он работает практически везде, и им каждый день пользуются миллионы людей, причем многие из них даже не подозревают об этом.

На bash повлияли многие проекты, начиная с седьмой редакции оригинальной командной оболочки Unix, написанной Стивеном Борном (Stephen Bourne). Наиболее существенное влияние оказал стандарт Posix, в котором определена значительная часть функциональных возможностей bash. Подобное сочетание обратной совместимости и соответствие стандартам добавило свои собственные проблемы.

Bash получает преимущество от того, что является частью проекта GNU, определяющего направление развития и границы, в которых существует bash. Без проекта GNU, не было бы никакого bash. Bash также выигрывает благодаря тому, что у него есть активное быстро реагирующее сообщество пользователей. Их отзывы помогли сделать bash тем, чем сегодня он стал, что свидетельствует о преимуществах свободного программного обеспечения.

Примечания:

1. Чаще всего — последовательность, состоящая из одного из таких повторяющихся символов.
2. Встроенная команда `exec` является исключением из этого правила.

Creative Commons

Перевод был сделан в соответствие с лицензией [Creative Commons](#). С русским вариантом лицензии можно ознакомиться [здесь](#).

4.Архитектура приложений с открытым исходным кодом. Том 1. Глава 4. Berkeley DB

Закон Конвея утверждает, что конструкция системы отражает структуру организации, которая ее создала. Если это утверждение слегка развить, то мы могли бы ожидать, что некий программный объект, спроектированный и первоначально созданный двумя разработчиками, должен был каким-то образом отражать, но не структуру организации, а внутренние предубеждения и философию каждого участника разработки. Один из авторов настоящей главы (Margo Seltzer) провела свою карьеру среди мировых файловых систем и систем управления базами данных. Если ее спросить, то она аргументировано докажет, что оба вида этих систем, по существу, являются одним и тем же,

и, более того, операционные системы и системы управления базами данных являются, по существу, менеджерами ресурсов и провайдерами удобных абстракций. Различия «всего лишь» в деталях реализации. Второй автор (Keith Bostic) верит в инструментальные подходы в инженерии программного обеспечения и в разработке компонентов на базе простых составных блоков, поскольку такие системы заведомо превосходят системы с монолитной архитектурой в части разных важных «особенностей»: понятности, расширяемости, поддерживаемости, тестируемости и гибкости.

Если объединить оба этих подхода, вы не удивитесь, узнав, что большую часть последних двух десятилетий мы потратили на совместную разработку Berkeley DB - библиотеки программ, обеспечивающих быстрое, гибкое, надежное и масштабируемое управление данными. Berkeley DB предоставляет практически те же самые функции, которые многие ожидают от более традиционных систем, например, реляционных баз данных, но они собраны по-другому. Например, в Berkeley DB имеется быстрый доступ к данным по ключам и последовательный доступ, и есть поддержка транзакций и восстановления после сбоя. Однако, эти функции не выделяются в отдельный сервер приложений, а реализованы в виде библиотеки, которая компонуется непосредственно с тем приложением, в котором нужен этот сервис.

В этой главе мы заглянем глубже в систему Berkeley DB и увидим, что она состоит из набора модулей, каждый из которых воплощает в себе философию Unix «хорошо делать что-то одно». Приложения, в которые включены компоненты Berkeley DB, могут обращаться к ним непосредственно, либо могут просто пользоваться ими неявно через более привычные операции получения, сохранения или удаления элементов данных. Мы сосредоточимся на архитектуре системы – как мы начали, что мы разрабатывали, где мы оказались и почему. Конструктивные особенности могут потребовать (и, безусловно, потребуют!) производить адаптацию и изменения – что с течением времени становится жизненно важным для поддержки системы и сохранения ее непротиворечивости. Мы также вкратце рассмотрим эволюцию кода длительно существующих программных проектов. Разработка Berkeley DB велась в течение более двух десятилетий, а это, неизбежно предполагает, что в ее основе лежал хороший проект.

4.1. В начале

Berkeley DB восходит к эпохе, когда операционная система Unix была собственностью компании AT&T и были сотни утилит и библиотек, в родословных которых присутствовали строгие лицензионные ограничения. Марго Зельцер (Margo Seltzer) была аспирантом Калифорнийского университета в Беркли, а Кейт Бостик (Keith Bostic) был членом группы Computer Systems Research Group. В то время Кейт работал над удалением проприетарного программного обеспечения AT&T из дистрибутива Berkeley Software.

Проект Berkeley DB начался со скромной задачи замены хэш-пакета hsearch, работающего в памяти, и хэш-пакетов dbm/ndbm, работающих с диском, новыми пакетами, имеющими лучшую реализацию хэш-функций, работающими как в памяти, так и с диском, и свободно распространяемыми без проприетарной лицензии. В основе библиотеки hash, написанной Марго Зельцер [SY91], лежало исследование линейно расширяемых хэш-таблиц, выполненное Витольдом Литвином (Witold Litwin). Библиотека была уникальна своей искусственной схемой, позволяющей получать константное время отображения между хэш-значениями и адресами страниц, а также своей возможностью обрабатывать большие объемы данных — элементы, размер которых больше размера используемых для этого хэш-блоков или страниц файловой системы, размер которых обычно составляет от четырех до восьми килобайтов.

Если хэш-таблицы — это хорошо, то деревья Btrees и хэш-таблицы — это еще лучше. Майк Олсон (Mike Olson), также аспирант Калифорнийского университета в Беркли, написал ряд реализаций деревьев Btree и согласился написать еще одну. Мы встроим преобразовали программу, созданную Марго для хэш-таблиц, и программу для работы с деревьями Btree, созданную Майком, в интерфейс API, не зависящий от метода доступа, с помощью которого приложения обращались к хэш-

таблицам или деревьям Btrees через специальные средства работы с базами данных, в которых были методы обработки, позволяющие читать и модифицировать данные.

Создавая эти два метода доступа, Майк Олсон и Марго Зельцер опубликовали научную статью ([\[SO92\]](#)), описывающую LIBTP, транзакционную библиотеку программного уровня, которая работает в адресном пространстве приложений.

Библиотеки хэш-таблиц и деревьев Btree были включены в окончательные релизы 4BSD под названием Berkeley DB 1.85. Технически, в методе доступа Btree были реализованы деревья вида B+link (двоичные деревья + ссылки), однако, в оставшейся части настоящей статьи мы будем пользоваться термином Btree, поскольку это название метода доступа. Вероятно, каждому, кто пользовался какой-нибудь системой, базирующейся на Linux или BSD, знакомы структура и интерфейсы API Berkeley DB 1.85.

Библиотека Berkeley DB 1.85 не изменялась в течение нескольких лет до тех пор, пока в 1996 году компания Netscape не заключила с Марго Зельцер и Кейт Бостик контракт на разработку полностью транзакционной схемы, описанной в статье о LIBTP, и на создание версии приложения промышленного уровня. В результате этих усилий была создана первая транзакционная версия Berkeley DB - версия 2.0.

Последующая история Berkeley DB проще и более традиционна: в Berkeley DB 2.0 (1997) были предложены транзакционные операции для Berkeley DB; Berkeley DB 3.0 (1999) была вновь версией, которую перепроектировали и в которую были добавлены дополнительные уровни абстракции, что косвенно привело к росту функциональных возможностей. В Berkeley DB 4.0 (2001) была представлена репликация и реализована высокая степень готовности, а в Oracle Berkeley DB 5.0 (2010) была добавлена поддержка SQL.

К моменту написания данной главы Berkeley DB является самым широко используемой в мире инструментальным набором, предназначенным для работы с базами данных, с сотнями миллионов установленных копий, работающих на всем, начиная от маршрутизаторов и браузеров и до почтовых программ и операционных систем. Несмотря на то, что проекту более двадцати лет, базовый инструментальный набор и объектно-ориентированный подход, используемый в Berkeley DB, позволяют поэтапно улучшать и перерабатывать проект в соответствие с требованиями программ, в которых он используется.

Первый урок конструирования

Для тестирования и сопровождения любого сложного программного пакета жизненно важно, чтобы он был спроектирован и собран в виде набора взаимодействующих модулей с хорошо определенными границами действия API. Границы могут (и должны!) сдвигаться по мере необходимости, но они всегда должны в нем присутствовать. Существование этих границ не позволит программному пакету превратиться в клубок спагетти, который невозможно будет поддерживать. Батлер Лампсон однажды сказал, что все проблемы в области компьютерных наук можно решить с помощью еще одного уровня косвенности. Если конкретно, то когда его спросили, что это значит с точки зрения объектной ориентированности, Лампсон ответил, что это означает, что за API могут быть несколько реализаций. Этот подход, позволяющий иметь за единым интерфейсом несколько реализаций, был воплощен в общей схеме и в реализации Berkeley DB, что позволило реализовать объектно-ориентированную систему несмотря на то, что библиотека написана на языке С.

4.2. Обзор архитектуры

В этом разделе мы рассмотрим архитектуру библиотеки Berkeley DB, причем начнем с LIBTP и укажем ключевые аспекты ее эволюции.

На рис.4.1, который взят из оригинальной статьи Зельцер и Олсона, показана первоначальная архитектура LIBTP, а на рис.4.2 приведена архитектура Berkeley DB 2.0.

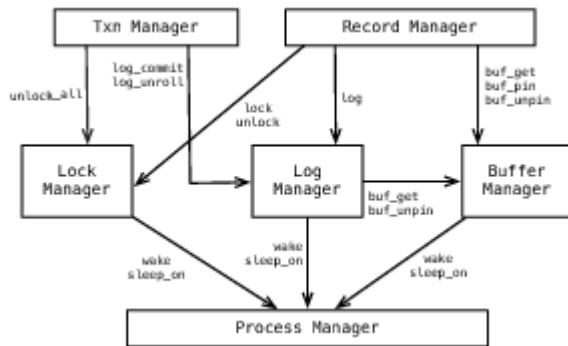


Рис.4.1: Архитектура прототипной системы LIBTP

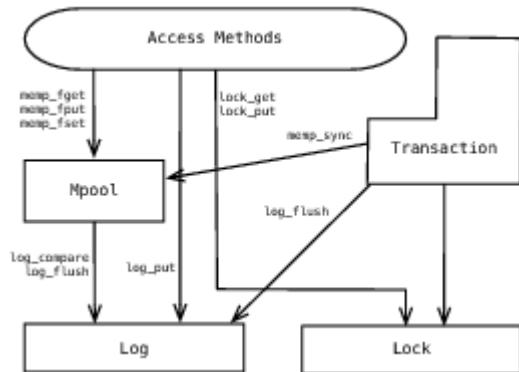


Рис.4.2: Архитектура Berkeley DB-2.0, которую предполагалось реализовать

Единственным существенным различием между реализацией LIBTP и архитектурой Berkeley DB 2.0 было удаление менеджера процессов. Вместо того, чтобы использовать синхронизацию на уровне подсистемы, в LIBTP требовалось, чтобы каждый поток управления регистрировал себя в библиотеке, а затем синхронизировался с отдельными потоками / процессами. Как будет рассказано в разделе 4.4, первоначальная архитектура, возможно, была для нас более предпочтительной.

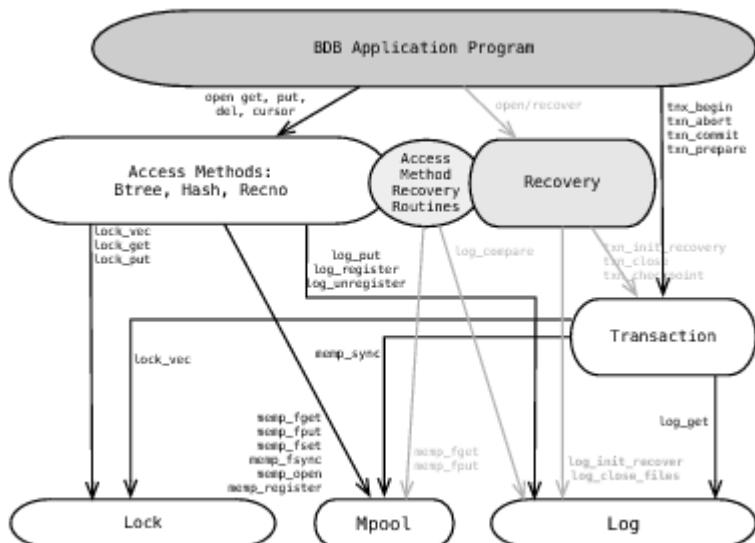


Рис.4.3: Архитектура Berkeley DB 2.0.6, которая была фактически реализована

Различие между проектом и фактической реализацией архитектуры DB-2.0.6, приведенной на рис.4.3, иллюстрирует реальность реализации надежного менеджера восстановлений. Подсистема восстановления показана серым цветом. В ее состав входят как инфраструктура драйверов, наход-

дящаяся в блоке «Recovery» («Восстановление»), так и и набор процедур redo и undo, осуществляющих восстановление после операций, выполняемых методами доступа. Они представлены в овале с надписью «Access method recovery routines» («Процедуры восстановления действий, выполняемых методами доступа»). В Berkeley DB 2.0 есть единая схема, используемая при восстановлении, в отличие от жестко закодированных процедур журналирования и восстановления для конкретных методов доступа, что было сделано в LIBTP. Такая универсальная схема также позволила создать более богатый интерфейс между различными модулями.

На рис.4.4 проиллюстрирована архитектура Berkeley DB-5.0.21. Цифры на диаграмме указывают интерфейсы API, перечисленные в таблице 4.1. Хотя первоначальная архитектура все еще просматривается, в нынешней архитектуре видно ее развитие благодаря тому, что добавлены новые модули, проведена декомпозиция старых модулей (например, модуль log был преобразован в модули log и dbreg), а также существенно увеличено количество межмодульных API.

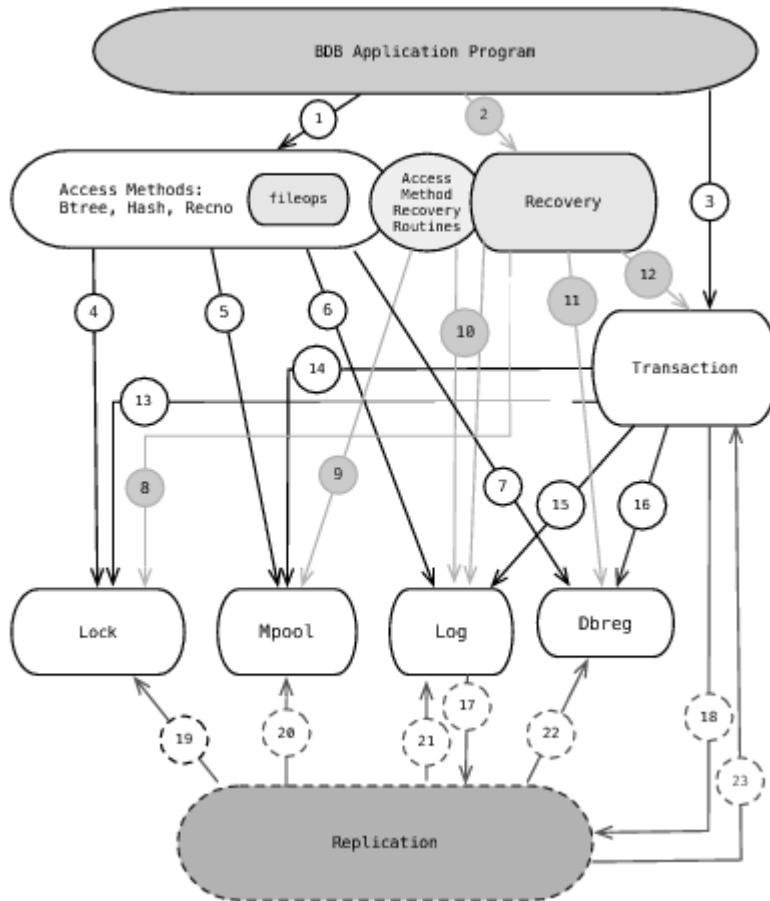


Рис.4.4: Архитектура Berkeley DB-5.0.21

Таблица 4.1: Интерфейсы API в Berkeley DB 5.0.21

Интерфейсы API для приложений

1. Операции обработки DBP

open
get
put
del

2. Восстановление DB_ENV

open(... DB_RECOVER ...)

3. Транзакционные интерфейсы API

DB_ENV->txn_begin
DB_TXN->abort
DB_TXN->commit
DB_TXN->prepare

cursor

Интерфейсы API, используемые в методах доступа

4. Into Lock	5. Into Mpool	6. Into Log	7. Into Dbreg
<code>_lock_downgrade</code>	<code>_memp_nameop</code>	<code>_log_print_record</code>	<code>_dbreg_setup</code>
<code>_lock_vec</code>	<code>_memp_fget</code>		<code>_dbreg_net_id</code>
<code>_lock_get</code>	<code>_memp_fput</code>		<code>_dbreg_revoke</code>
<code>_lock_put</code>	<code>_memp_fset</code>		<code>_dbreg_teardown</code>
	<code>_memp_fsync</code>		<code>_dbreg_close_id</code>
	<code>_memp_fopen</code>		<code>_dbreg_log_id</code>
	<code>_memp_fclose</code>		
	<code>_memp_ftruncate</code>		
	<code>_memp_extend_freeリスト</code>		

Интерфейсы API процесса восстановления

8. Into Lock	9. Into Mpool	10. Into Log	11. Into Dbreg	12. Into Txn
<code>_lock_getlocker</code>	<code>_memp_fget</code>	<code>_log_compare</code>	<code>_dbreg_close_files</code>	<code>_txn_getckpt</code>
<code>_lock_get_list</code>	<code>_memp_fput</code>	<code>_log_open</code>	<code>_dbreg_mark_restore</code>	<code>_txn_checkpoint</code>
	<code>_memp_fset</code>	<code>_log_earliest</code>	<code>_dbreg_init_recover</code>	<code>_txn_reset</code>
	<code>_memp_nameop</code>	<code>_log_backup</code>		<code>_txn_recycle_id</code>
		<code>_log_cursor</code>		<code>_txn_findlastckpt</code>
		<code>_log_vtruncate</code>		<code>_txn_ckp_read</code>

Интерфейсы API, используемые модулем транзакций

13. Into Lock	14. Into Mpool	15. Into Log	16. Into Dbreg
<code>_lock_vec</code>	<code>_memp_sync</code>	<code>_log_cursor</code>	<code>_dbreg_invalidate_files</code>
<code>_lock_downgrade</code>	<code>_memp_nameop</code>	<code>_log_current_lsn</code>	<code>_dbreg_close_files</code>
			<code>_dbreg_log_files</code>

Интерфейс API Into системы репликации

17. From Log	18. From Txn
<code>_rep_send_message</code>	<code>_rep_lease_check</code>
<code>_rep_bulk_message</code>	<code>_rep_txn_applied</code>
	<code>_rep_send_message</code>

Интерфейс API From системы репликации

19. Into Lock	20. Into Mpool	21. Into Log	22. Into Dbreg	23. Into Txn
<code>_lock_vec</code>	<code>_memp_fclose</code>	<code>_log_get_stable_l1</code>	<code>_dbreg_mark_restore</code>	<code>_txn_recycle_id</code>

	sn	d
<code>__lock_get</code>	<code>__memp_fget</code>	<code>__log_cursor</code>
<code>__lock_id</code>	<code>__memp_fput</code>	<code>__log_newfile</code>
	<code>__memp_fsync</code>	<code>__log_flush</code>
		<code>__log_rep_put</code>
		<code>__log_zero</code>
		<code>__log_vtruncate</code>
		<code>__dbreg_invalidate_files</code>
		<code>__txn_begin</code>
		<code>__dbreg_close_files</code>
		<code>__txn_recover</code>
		<code>__txn_getckpt</code>
		<code>__txn_updateckpt</code>

Таблица 4.1: Интерфейсы API в Berkeley DB 5.0.21

Спустя более десяти лет эволюции, десятков коммерческих релизов, а также добавления сотен новых функций мы видим, что архитектура стала существенно более сложной, чем ее предшественница. Ключевые моменты, которые необходимо отметить, следующие: во-первых, репликация добавила в систему совершенно новый слой, но это сделано аккуратно, так что взаимодействие с остальной частью системы осуществляется через те же самые API, что и исторически сложившийся код. Во-вторых, модуль `log` был разделен на модули `log` и `dbreg` (регистрация базы данных). Этот вопрос будет более подробно рассмотрен в разделе 4.8. В-третьих, для того, чтобы в приложениях не было коллизий с именами наших функций, мы поместили все межмодульные вызовы в пространство имен, в котором в именах используется предваряющий символ подчеркивания. Мы обсудим этот вопрос в Шестом уроке конструирования.

В-четвертых, сейчас интерфейс API подсистемы журналирования базируется на использовании курсоров (это не интерфейс API `log_get`, он заменен интерфейсом API `log_cursor`). Исторически сложилось так, что Berkeley DB никогда в любой момент времени не использовалось более одного потока чтения или записи журнала, поэтому в библиотеке имелось только одно понятие текущего указателя поиска в журнале. С точки зрения абстрагирования это никогда не считалось хорошим решением, но при наличии репликации оно стало неприемлемым. Точно также, как в интерфейсе приложений API поддерживается итерация с использованием курсоров, так и в журнале теперь итерация выполняется с применением курсоров. В-пятых, модуль `fileop`, входящий в состав модулей доступа, обеспечивает в транзакционно защищенной базе данных поддержку операций создания, удаления и переименования. Нам потребовалось много попыток с тем, чтобы сделать реализацию приемлемой (она еще не так аккуратна, как этого бы хотелось), а после переделок ее в течение длительного времени мы выделили ее в отдельный модуль.

Второй урок конструирования

Программный проект является просто одним из нескольких способов, которые заставляют вас подумать о всей проблеме в целом прежде, чем вы ее пытаетесь решить. Опытные программисты для этого используют различные методики: некоторые пишут первую версию и выбрасывают ее, некоторые пишут множество страниц руководств или проектной документации, другие заполняют некоторый шаблон, в котором каждое требование определяется, назначается определенной функции или комментируется. Например, в Berkeley DB, мы прежде, чем писать какой-либо код, создавали полный набор справочных страниц в стиле Unix для всех методов доступа и для всех компонентов, лежащих в их основе. Вне зависимости от используемой методики, трудно ясно разобраться в архитектуре программы после того, как начинается отладка кода, не говоря уже о том, что трудно делать крупные изменения в архитектуре, при которых часто зря пропадают усилия, ранее затраченные на отладку. Создание архитектуры программы требует другого склада ума, отличающегося от того, который нужен при отладке кода, и архитектура, которая у вас есть к моменту, когда вы начинаете отладку, как правило, именно та, которую вы реализовали в конкретной версии.

Почему вся архитектура библиотека транзакций собрана из компонентов, а не заточена на один из вариантов предполагаемого использования? На этот вопрос есть три ответа. Во-первых, это повыш-

шает дисциплину проектирования. Во-вторых, без четко определенных границ в коде сложные программные пакеты неизбежно выродятся в груду кода, который невозможно будет поддерживать. В-третьих, вы никогда не можете предвидеть все способы использования клиентами вашего программного обеспечения; если вы предоставите пользователям больше возможностей, позволив им получать доступ к компонентам, они будут пользоваться ими так, как вы никогда на это не рассчитывали.

В следующих разделах мы рассмотрим все компоненты Berkeley DB, разберемся, что они делают и как они вписываются в общую картину.

4.3. Методы доступа: Btree, Hash, Recno, Queue

Методы доступа в Berkeley DB позволяют осуществлять поиск как по ключу, так и методом итерации, и использовать байтовые строки переменной и фиксированной длины. В методах доступа BTee и Hash поддерживается использование пар ключ/значение с переменной длиной строк. В Recno и Queue поддерживается использование пар номер-запись/значение (причем Recno поддерживает использование значений переменной длины, а Queue поддерживает значения только фиксированной длины).

Основное различие между методами доступа BTee и Hash в том, что в BTee для ключей предлагается локализация ссылок тогда, как в Hash - нет. Это означает, что BTee является правильным методом доступа почти для всех наборов данных; но метод доступа Hash пригоден для таких больших наборов данных, для которых даже структура индексов BTee не поместится в память. В таком случае память лучше использовать для данных, а не для структур индексов. Этот компромисс имел гораздо больше смысла в 1990 году, когда оперативной памяти было обычно намного меньше, чем сегодня.

Разница между методами Recno и Queue в том, что в Queue поддерживается блокировка на уровне записей, из-за чего можно использовать значения только фиксированной длины. В Recno можно использовать объекты переменной длины, но, как и в случае с BTee и Hash, поддерживается блокировка записей только на уровне страниц.

Первоначально мы разрабатывали Berkeley DB таким образом, чтобы функции CRUD (функции Create - создание, Read - чтение, Update - обновление и Delete - удаление) использовали ключи, чтобы приложениям был предоставлен простейший интерфейс. Впоследствии для поддержки итерации мы добавили курсоры. Такая последовательность разработки привела к беспорядку и расточительным случаям, когда внутри библиотеки дублировались большие фрагменты путей исполнения кода. Со временем это привело к невозможности осуществлять поддержку и мы преобразовали все операции с ключами в операции с курсорами (операции с использованием ключей теперь вызывают из кэша курсор, выполняют операцию и возвращают курсор в пул курсоров). Это один из примеров бесконечно повторяемых правил разработки программного обеспечения: не оптимизировать порядок исполнения кода, если это ухудшает его понятность и увеличивает сложность, до тех пор, пока вы не будете знать, что необходимо сделать именно так.

Третий урок конструирования

Архитектура программы ухудшается не из-за того, что она постепенно стареет. Архитектура программы ухудшается прямо пропорционально количеству вносимых в программу изменений: исправление ошибок ухудшает деление на слои, а новые возможности оказывают воздействие на всю конструкцию. Решение о том, что архитектура программы ухудшилась настолько, что вы должны изменить всю конструкцию или переписать модуль, является тяжелым решением. С одной стороны, поскольку архитектура ухудшилась, сопровождение и дальнейшее разработка программы становятся все труднее, и в конце концов сопровождение устаревшего фрагмента можно поддерживать в каждом релизе только с привлечением грубой силы тестировщиков, поскольку никто

не понимает, как программа работает внутри. С другой стороны, пользователи будут горько жаловаться на нестабильность и несовместимости, которые являются следствием фундаментальных изменений. Независимо от того, какой путь вы выбирете, вы, как архитектор программы, можете лишь гарантировать, что кто-нибудь будет недоволен.

Мы опускаем подробное обсуждение внутренних особенностей методов доступа в Berkeley DB; они реализуют довольно известные алгоритмы Btree и хэширования (Recno является слоем над кодом Btree, а Queue является функцией поиска файловых блоков, хотя и усложненной за счет добавления блокировок на уровне записей).

4.4. Интерфейсный слой библиотеки

Со временем, когда мы добавили дополнительные функциональные возможности, мы обнаружили, что как для приложений, так и для внутреннего кода нужны одни и те же функции высокого уровня (например, для выполнения операции объединения таблиц необходимо несколько курсоров для итерации по строкам точно также, как и в приложение может использоваться курсор для итерации по тем же самим строкам).

Четвертый урок конструирования

Не имеет значения, как вы называете свои переменные, методы, функции, что комментируете или какой стиль кодирования вы используете; то есть, существует большое количество форматов и стилей, которые считаются «достаточно хорошими». Что действительно важно, причем имеет большое значение, это чтобы имена и стиль были согласованными. Опытные программисты получают массу сведений из текста кода и именования объектов. Вы должны следить за тем, чтобы не было несоответствия между именами и стилем, поскольку некоторые программисты тратят время и усилия на то, чтобы не передавать другим верные сведения, и наоборот. Несоблюдение принятых соглашений о кодировании является тяжким преступлением.

По этой причине, мы разбили интерфейсы API методов доступа на строго определенные слои. В этих слоях, содержащих интерфейсные процедуры, выполняются все необходимые общие проверки ошибок, проверки ошибок конкретных функций, отслеживается работа интерфейса, а также выполняются другие задачи, такие как автоматическое управление транзакциями. Когда приложения обращаются к Berkeley DB, они обращаются к интерфейсным процедурам первого уровня, использующим методы работы с дескрипторами объектов. (Например, `__dbc_put_pp` является обращением к интерфейсу вызова метода `«put»`, использующего курсор Berkeley DB, с помощью которого обновляется элемент данных. `«_pp»` является суффиксом, используемым нами для идентификации всех функций, к которым могут обращаться приложения).

Одной из задач Беркли DB, которые выполняются в интерфейсном слое, является отслеживание запуска потоков исполнения внутри библиотеки Berkeley DB. Это необходимо потому, что некоторые внутренние операции Berkeley DB можно выполнять только тогда, когда ни один из потоков не работает внутри библиотеки. Отслеживание работы потоков в библиотеке Berkeley DB реализуется с помощью установки флага, указывающего, что внутри библиотеки был запущен поток, в начале каждого библиотечного API и сброса этого флага в момент, когда происходит выход из вызова этого API. Такая проверка входа/выхода всегда выполняется в интерфейсном слое, поскольку она эквивалентна проверке, определяющей, выполнен ли вызов в реплицируемой среде.

Возникает естественный вопрос: «а почему бы не передать идентификатор потока внутрь библиотеки, не будет ли это проще?» Ответ – да, было бы гораздо проще, и мы, конечно, хотели сделать именно так. Но такое изменение потребовало бы изменить каждое отдельное приложение Berkeley DB и большинство обращений каждого из приложений к Berkeley DB, а во многих случаях также потребовалось бы переделать структуру приложений.

Пятый урок конструирования

Архитекторы программ должны аккуратно относиться к обновлениям в реально используемых проектах: пользователи воспримут нормально незначительные изменения, осуществляемые при обновлениях до новых версий (если вы гарантируете отсутствие ошибок времени компиляции, то есть отсутствие очевидных отказов до тех пор, пока не будет завершено обновление; изменения при обновлении никогда не должны приводить к плохо диагностируемым отказам). Но чтобы сделать по-настоящему фундаментальные изменения, вы должны решиться на использование нового кода базы данных и потребовать от ваших пользователей портировать их базы данных. Очевидно, что новый код и портирование являются не дешевыми с точки зрения затраты времени или ресурсов, но никто из ваших пользователей не рассердится, если вы расскажете им, что огромный капитальный ремонт является в действительности незначительным обновлением.

Другой задачей, выполняемой в интерфейсном слое, является создание транзакций. Библиотека Berkeley DB поддерживает режим, в котором каждая операция выполняется в рамках автоматически создаваемой транзакции (это освобождает приложение от создания и подтверждение своих собственных явно указываемых транзакций). Для поддержки этого режима требуется, чтобы транзакция создавалась автоматически каждый раз, когда приложение осуществляет обращение через API и не указывает, что оно использует свои собственные транзакции.

Наконец, во всех интерфейсах API в Berkeley DB API необходима проверка аргументов. В Berkeley DB есть два вида проверки ошибок - общая проверка, которая определяет, была ли наша база данных повреждена во время предыдущей операции и не находимся ли мы в процессе изменения реплицируемого состояния (например, выполняются изменения, при которых можно писать реплики). Есть также конкретные проверки для интерфейса API: правильность использования флага, правильность использования параметров, правильность комбинации параметров, а также отсутствие любых других видов ошибок, что мы должны поверить перед тем, как действительно будем выполнять требуемую операцию.

Эти проверки, необходимые для определенных API, инкапсулированных внутри функций, имеющих суффикс `_arg`. Таким образом, проверка ошибок для метода `put`, используемого с курсором, находится в функции `__dbc_put_arg`, которая вызывается из функции `__dbc_put_pp`.

Наконец, когда будут завершены все проверки аргументов и будет создана транзакция, мы вызываем рабочий метод, который фактически выполняет операцию (в нашем примере это будет `__dbc_put`), которая будет той же самой функцией, которую мы используем, когда обращаемся внутри библиотеки к функции `put` курсора.

Эта декомпозиция была создана в период интенсивной деятельности, когда мы пытались точно определить, какие действия мы должны выполнять при работе в реплицируемых средах. После того, как мы некоторое достаточно большое количество раз повторно просмотрели код, мы выделили всю эту предварительную проверку отдельно с тем, чтобы следующий раз, когда мы обнаружим, что проблема в ней, ее было проще изменять.

4.5. Компоненты, лежащие глубже

Есть четыре компонента, лежащие в основе методов доступа: менеджер буферирования, менеджер блокировок, журнальный менеджер и менеджер транзакций. Мы обсудим каждый из них в отдельности, но у них всех есть некоторые общие архитектурные особенности.

Во-первых, во всех подсистемах имеются свои собственные API, и первоначально в каждой подсистеме имелся свой собственный дескриптор объектов со всеми методами для конкретной подсистемы, использовавший этот дескриптор. Например, вы могли использовать менеджер блокировок из Berkeley DB для обработки ваших собственных блокировок или могли написать свой собст-

венный отдельный менеджер блокировок, либо вы могли использовать менеджер буферирования из Berkeley DB для того, чтобы обрабатывать страницы ваших собственных файлов в разделяемой памяти. Со временем для того, чтобы упростить приложения Berkeley DB, дескрипторы, предназначенные для использования в конкретных подсистемах, были удалены из интерфейса API. Хотя подсистемы все еще являются отдельными компонентами, которые можно использовать независимо от других подсистем, теперь у них всех есть общее средство управления объектами — средство управления "средой" `DB_ENV`. Эта архитектурная особенность позволяет выделять слои и делать обобщения. Даже несмотря на то, что время от времени слой меняет свое положение, и все еще есть несколько мест, где к некоторой подсистеме можно получить доступ через другую подсистему, программисты могут рассматривать каждую часть системы как отдельный программный продукт с его собственными возможностями.

Во-вторых, все подсистемы (на самом деле, все функции Berkeley DB) возвращают коды ошибок через стек вызовов. Berkeley DB, как библиотека, не может для объявления глобальных переменных пользоваться пространством имен приложения, не говоря уже о том, что возврат через стек вызовов в случае ошибок является хорошей дисциплиной для программиста

Шестой урок конструирования

Когда создается библиотека, жизненно важно не нарушать пространство имен. Программисты, использующие вашу библиотеку, не должны запоминать десятки зарезервированных имен функций, констант, структур и глобальных переменных для того, чтобы избежать конфликтов имен между приложением и библиотекой.

Наконец, все подсистемы поддерживают использование разделяемой памяти. Поскольку в Berkeley DB поддерживается разделение баз данных между несколькими работающими процессами, все интересуемые структуры данных должны находиться в разделяемой памяти. Наиболее значимым следствием такого решения является то, что для того, чтобы структуры данных, использующие указатели, работали в контексте нескольких процессов, в структурах данных, располагаемых в памяти, должны использоваться пары базовый адрес и смещение, а не указатели. Другими словами, вместо косвенного доступа через указатель, библиотека Berkeley DB должна создавать указатель из базового адреса (адреса, по которому сегмент разделяемой памяти отображается в память) и смещения (смещение конкретной структуры данных в соответствие с тем, как она отображена в сегменте). Для поддержки этой возможности мы написали версию пакета `queue` для дистрибутива Berkeley Software, в котором был реализован широкий спектр различных связанных списков.

Седьмой урок конструирования

Еще перед тем, как мы написали пакет связных списков для разделяемой памяти, инженеры Berkeley DB вручную кодировали разнообразные варианты различных структур данных, используемых в разделяемой памяти, и эти реализации рассыпались и их было трудно отлаживать. Пакет со списками для разделяемой памяти, сделанный наподобие пакетов списков BSD (`queue.h`), заменил все эти усилия. Как только пакет был отложен, нам больше не потребовалось выполнять какие-либо другие отладки, касающиеся проблем связных списков в разделяемой памяти. Это иллюстрирует три важных принципа проектирования: Во-первых, если у вас есть функциональные фрагменты, которые появляются более одного раза, напишите разделяемые функции и используйте их, поскольку само по себе наличие в коде двух копий любых конкретных функциональных фрагментов гарантирует, что один из них будет реализован неправильно. Во-вторых, когда вы разрабатываете набор процедур общего назначения, напишите набор тестов для этого набора процедур для того, чтобы вы могли отладить их отдельно. В-третьих, чем труднее писать код, тем более важно чтобы он был написан и поддерживался отдельно; почти невозможно защитить окружающий код от инфицирования и разъедания со стороны некоторого фрагмента кода.

4.6. Менеджер буферирования: Mpool

Подсистема Mpool в Berkeley DB является размещаемым в памяти пулом буферов файловых страниц, скрывающим тот факт, что основная память является ограниченным ресурсом, для которого требуется библиотека, осуществляющая при работе с базами данных, размер которых больше размера памяти, перемещение страниц базы данных на диск и с диска в память. Кэширование страниц базы данных в памяти, было тем, что позволило исходной библиотеке хеширования значительно превзойти по производительности ранее использовавшиеся реализации `hsearch` и `ndbm`.

Хотя метод доступа Btree в Berkeley DB является достаточно традиционной реализацией B+tree, указатели между узлами дерева представлены в виде номеров страниц, а не в виде фактических указателей памяти, потому что реализация библиотеки использует формат данных, предназначенный для диска, также в качестве формата данных, используемого в памяти.

Есть и другие последствия влияния на производительность, обусловленные тем, что представление индексов Berkeley DB в памяти является на самом деле кэшем данных, постоянно хранящихся на диске. Например, всякий раз, когда Berkeley DB обращается к кэш-странице, она сначала фиксирует размещение страницы в памяти. Эта фиксация запрещает каким-либо другим потокам или процессам убирать эту страницу из пула буферов. Даже если индексная структура полностью помещается в кэш-памяти и ее не нужно сбрасывать на диск, Berkeley DB при каждом доступе, по-прежнему, сначала выполняет фиксацию страниц в памяти, а затем — их освобождение, поскольку согласно лежащей в основе модели, предлагаемой Mpool, это — кэш, а не постоянное хранилище данных.

4.6.1. Файловая абстракция Mpool

Предполагается, что Mpool находится на вершине файловой системы, экспортируя файловые абстракции через интерфейс API. Например, обработчики `DB_MPOOLFILE`, используемые для файлов, расположенных на диске, представляют методы для перемещения страниц в файл и выборки страниц из файла. Хотя в Berkeley DB поддерживаются временные базы данных и базы данных, размещаемые только в памяти, для них также используются обработчики `DB_MPOOLFILE` из-за лежащей в их основе абстракций Mpool. Методы `get` и `put` являются основными в интерфейсах Mpool API: метод `get` гарантирует, что страница размещена в кэше, закрепляет страницу в памяти и возвращает указатель на эту страницу. Когда библиотека выполнит все действия с этой страницей, метод `put` отменяет закрепление этой страницы, что позволит удалять страницу из памяти. В ранних версиях Berkeley DB не было различия между закреплением страницы для чтения и закреплением страницы для записи. Однако для того, чтобы увеличить степень распараллеливания, мы расширили интерфейс Mpool API с тем, чтобы при обращении можно было сообщить о намерении обновить страницу. Эта возможность различать доступ для чтения от доступа для записи, имеет важное значение для реализации многовариантного управления распараллеливанием. Страница, закрепленная в памяти для чтения, если случится, что она изменится, может быть записана на диск, тогда как страница, закрепленная для записи, не может, поскольку она в любой момент может находиться в несогласованном состоянии.

4.6.2. Запись в журнал с упреждением

В Berkeley DB в качестве транзакционного механизма, позволяющего выполнить восстановление после возможного отказа, используется запись в журнал с упреждением (или WAL - write-ahead-logging). Термин «запись в журнал с упреждением» определяет политику, требующую, чтобы записи в журнал, описывающие любые изменения, делались на диск *перед* фактическим обновлением данных, которые они описывают. Использование в Berkeley DB политики WAL в качестве транзакционного механизма имеет важные последствия для Mpool, и в Mpool приходится, с точки зрения проектирования, балансировать между реализацией общего механизма кэширования и необходимостью поддерживать протокол WAL.

В Berkeley DB на всех страницах данных записываются журнальные порядковые номера (номера LSN – log sequence number), позволяющие заносить в журнал записи, соответствующие самым последним обновлениям конкретной страницы. Чтобы реализовать политику WAL, нужно перед тем, как Mpool запишет какую-нибудь страницу на диск, удостовериться, что регистрационная запись, соответствующая номеру LSN на странице, сохранена на диске. Задача проектирования состоит в том, чтобы реализовать эту функциональность так, чтобы ото всех клиентских программ Mpool не требовалось использовать формат страниц, идентичный используемому в Berkeley DB. В Mpool эта задача решается с помощью набора методов `set` (и `get`), которые обеспечивают необходимое поведение. Метод `set_lsn_offset` из `DB_MPOOLFILE` указывает смещение в байтах, где на странице Mpool должен искать номер LSN, необходимый для WAL. Если обращений к этому методу не было, то протокол WAL в Mpool не используется. Аналогичным образом метод `set_clearlen` сообщает Mpool, сколько байтов на странице занимают метаданные, которые нужно явно очистить, когда страница создается в кеше. Эти интерфейсы API позволяют реализовать в Mpool функции, необходимые для поддержки требований транзакционности, имеющиеся в Berkeley DB, не заставляя от всех пользователей Mpool реализовывать эти возможности.

Восьмой урок конструирования

Запись в журнал с упреждением – это еще один пример применения инкапсуляции и выделения слоя, причем даже в случае, когда эти функциональные возможности никогда не будут использоваться в другой части программного обеспечения: в конце концов, в каком количестве программ задумываются о номерах LSN в кэше? Несмотря на это, дисциплина полезна и облегчает поддержку программного обеспечение, его тестирование, отладку и расширение.

4.7. Менеджер блокировок: Lock

Точно также, как и Mpool, менеджер блокировок был разработан как компонент общего назначения: иерархический менеджер блокировок (смотрите [[GLPT76](#)]) создан для поддержки иерархии объектов, которые могут быть заблокированы (например, отдельные элементы данных) – страница, на которой размещен элемент данных, файл, в котором запомнен элемент данных или даже набор файлов. Поскольку мы описываем возможности менеджера блокировок, мы также объясним, как они используются в Berkeley DB. Но, как и с Mpool, важно запомнить, что другие приложения могут использовать менеджер блокировок совершенно по-другому и это нормально - он был разработан максимально гибко и поддерживает множество различных вариантов применений.

В менеджере блокировок есть три ключевые абстракции: «блокировка» (`«locker»`), которая определяет, на действия какого объекта устанавливается блокировка, «объект блокировки» (`«lock_object»`), который определяет блокируемый элемент, и «матрица конфликтов» (`«conflict matrix»`).

Блокировки являются 32-разрядными целыми числами. Berkeley DB делит пространство имен 32-разрядных чисел на транзакционные и на нетранзакционные блокировки (хотя это различие является прозрачным для менеджера блокировок). Когда Berkeley DB использует менеджер блокировок, он назначает идентификаторы ID блокировок в диапазоне от 0 до 0x7fffffff для нетранзакционных блокировок и в диапазоне от 0x80000000 и до 0xffffffff — для транзакционных блокировок. Например, когда приложение открывает базу данных, Berkeley DB для того, чтобы обеспечить, чтобы никакой другой поток управления не удалил ее или не переименовал, пока она используется, устанавливает для этой базы долговременную блокировку чтения. Поскольку это долговременная блокировка, она не принадлежит какой-либо транзакции и объект "блокировка", осуществляющий эту блокировку, является нетранзакционным.

Необходимо, чтобы любое приложение, использующее менеджер блокировок, назначало идентификаторы блокировок, поэтому в интерфейсе API менеджера блокировок есть вызовы `DB_ENV->lock_id` и `DB_ENV->lock_id_free`, предназначенные для выделения и освобождения блокировок.

Таким образом, приложениям не нужно реализовывать свои собственные механизмы выделения блокировок, хотя они, конечно, могут это делать.

4.7.1. Блокируемые объекты

Блокируемые объекты являются строками байтов произвольной длины и с любой внутренней структурой, представляющие блокируемые объекты. Когда две различных блокировки хотят заблокировать некоторый конкретный объект, они для ссылки на этот объект пользуются одной и той же строкой байтов. То есть, обязанность самих приложений соблюдать конвенции, касающиеся внутренней структуры блокируемых объектов.

Например, Berkeley DB использует структуру DB_LOCK_ILOCK для описания блокировок баз данных. В этой структуре есть три поля: идентификатор файла, номер страницы и тип.

Почти во всех случаях, Berkeley DB нужно описывать только конкретный файл и страницу, которую она хочет заблокировать. Berkeley DB назначает уникальный 32-разрядный номер каждой базе данных во время ее создания, записывает его в страницу метаданных базы данных, а затем использует его в качестве уникального идентификатора базы данных в подсистеме Mpool, подсистеме блокировок и журнальной подсистеме. Это идентификатор `fileid`, к которому мы ссылаемся в структуре DB_LOCK_ILOCK. Естественно, что номер страницы указывает, какую страницу из определенной базы данных мы хотим заблокировать. Когда мы обращаемся к блокировке страниц, мы задаем значение в поле типа в структуре DB_PAGE_LOCK. Но мы можем при необходимости заблокировать также другие типы объектов. Как уже упоминалось ранее, мы иногда блокируем дескриптор базы данных, для которого нужно указать тип DB_HANDLE_LOCK. Тип DB_RECORD_LOCK позволяет нам в методе доступа к очереди выполнить блокировку уровня записи, а тип DB_DATABASE_LOCK позволяет нам заблокировать всю базу данных.

Девятый урок конструирования

Решение в Berkeley DB использовать блокировку на уровне страниц было сделано по уважительным причинам, но мы обнаружили, что иногда из-за этого решения возникают проблемы. Блокировка на уровне страниц ограничивает возможность распараллеливания приложений, поскольку один поток, модифицирующий запись на странице базы данных, будет мешать другим потокам управления модифицировать другие записи на той же самой странице, тогда как при блокировке на уровне записей такое распараллеливание допускается до тех пор, пока два потока управления не будут модифицировать одну и ту же запись. Блокировка на уровне страниц повышает стабильность базы данных, поскольку она ограничивает количество возможных путей восстановления (во время восстановления страница всегда находится в одном из двух состояний, в отличие от бесконечного числа возможных состояний, в которых страница может быть в случае, если на странице добавляется или удаляется несколько записей). Поскольку Berkeley DB предназначалась для использования в качестве встроенной системы, когда нет администратора базы данных, который мог бы исправить ситуацию в случае возникновения повреждения, мы отдали предпочтение стабильности, а не возможности увеличения распараллеливания.

4.7.2. Матрица конфликтов

Последней абстракцией подсистемы блокировок, которую мы обсудим, является матрица конфликтов. В матрице конфликтов определяются различные типы блокировок, присутствующие в системе и их взаимодействие между собой. Давайте назовем сущность, удерживающую блокировку, владельцем блокировки, а сущность, пытающуюся выполнить блокировку, инициатором блокировки, и давайте считать, что владелец и инициатор имеют разные идентификаторы блокировок. Матрица конфликтов представляет собой массив, индексируемый с помощью пары [инициатор] [владелец], где каждый элемент содержит ноль в случае, если конфликта нет, что указывает, что запрашиваемая блокировка может быть выполнена, и содержит единицу в случае, если имеется конфликт, что указывает, что запрос не может быть выполнен.

В менеджере блокировок имеется матрица конфликтов, используемая по умолчанию, в которой указано именно то, что требуется для Berkeley DB, однако, приложение может использовать свои собственные режимы блокировок и матрицу конфликтов, соответствующие собственным целям приложения. Единственное требование к матрице конфликтов – чтобы она была квадратная (в ней одинаковое количество строк и столбцов) и чтобы в приложении для описания его собственных режимов блокировок (например, чтения, записи и т. д.) использовались последовательно идущие целые числа, начинающиеся с нуля. В таблице 4.2 показана матрица конфликтов для Berkeley DB.

Владелец		No-Lock	Read	Write	Wait	iWrite	iRead	iRW	uRead	wasWrite
Read			✓		✓		✓		✓	
Write			✓	✓	✓	✓	✓	✓	✓	✓
Wait										
iWrite			✓	✓			✓		✓	
iRead				✓					✓	
iRW			✓	✓			✓		✓	
uRead			✓		✓		✓			
wasWrite			✓	✓		✓	✓	✓		✓

Таблица 4.2: Матрица конфликтов чтения-записи

4.7.3. Поддержка иерархической блокировки

Прежде чем объяснять различные режимы блокировок, указываемые в матрице конфликтов Berkeley DB, давайте поговорим о том, как в подсистеме блокировок поддерживаются иерархические блокировки. Иерархическая блокировка представляет собой возможность блокировать различные элементы в иерархии вложения. Например, файлы содержат страницы, а страницы содержат отдельные элементы. Когда в иерархической системе блокировок изменяется один элемент на странице, нам нужно блокировать только этот элемент; если мы изменили все элементы страницы, было бы более эффективным просто заблокировать страницу, и если мы изменили все страницы файла, было бы лучше заблокировать весь файл. Кроме того, иерархическая блокировка должна различать иерархию контейнеров, поскольку блокировка страницы также говорит о блокировке файлов: вы не можете изменить файл, содержащий страницу, в тот же самый момент, когда модифицируются страницы файла.

Тогда вопрос состоит в том, как разрешать различным блокировщикам выполнять блокировку на различных иерархических уровнях, но при этом не получить в результате хаос. Ответ кроется в конструкции, которая называется уведомлением о блокировке (intention lock). Блокировщик создает внутри контейнера уведомление о блокировке с тем, чтобы сообщить о намерении заблокировать объекты внутри этого контейнера. Поэтому получение блокировки чтения на странице подразумевает получение уведомления о блокировке файла. Аналогично, чтобы записать единственный элемент страницы, вы должны создать уведомление о блокировке записи, как для страницы, так и для файла. В матрице конфликтов, приведенной выше, блокировки iRead, iWrite и iWR являются уведомлениями о блокировках, которые указывают на намерение выполнить чтение, запись или и то и другое, соответственно.

Поэтому когда выполняется иерархическая блокировка, а не запрос на отдельную блокировку, необходимо запрашивать потенциально много блокировок: блокировку на фактически блокируемый объект, а также уведомления о блокировках на любой объект, в котором содержится блокируемый

объект. В результате в Berkeley DB нужно обращаться к интерфейсу `DB_ENV->lock_vec`, который получает массив запросов на блокировку и атомарно реализует их (или отклоняет).

Хотя внутри самой Berkeley DB иерархическая блокировка не используется, она дает преимущество за счет возможности указывать различные матрицы конфликтов и возможности указывать за один раз сразу несколько запросов. Когда поддерживаются транзакции, мы используем матрицу конфликтов, предлагаемую по умолчанию, но с помощью другой матрицы конфликтов можно реализовать простой одновременный доступ без поддержки транзакций и восстановления. Чтобы подключить блокировки, мы пользуемся `DB_ENV->lock_vec`, методикой, которая улучшает распараллеливание при обходе дерева Btree [[Com79](#)]. Когда вы подключаете блокировку, вы ее сохраняете только в течение того времени, которое необходимо для получения следующей блокировки. То есть, вы блокируете внутреннюю страницу Btree только в течение того времени, которое нужно для чтения информации, позволяющей вам выбрать и заблокировать страницу на следующем уровне.

Десятый урок конструирования

Универсальная архитектура Berkeley DB принесла хорошие плоды, когда мы добавили функции параллельной работы с хранилищами данных. Первоначально в Berkeley DB предлагалось только два режима: либо вы, когда записывали данные, работали без какого-либо распараллеливания, либо - с полной поддержкой транзакций. Поддержка транзакций влечет за собой определенные сложности для разработчиков, и мы выяснили, что для некоторых приложений требовалась большая степень распараллеливания без накладных расходов, связанных с полной поддержкой транзакций. Чтобы реализовать эту возможность, мы добавили поддержку блокировок на уровне API, что позволяет пользоваться распараллеливанием при одновременной гарантии отсутствия тупиковых ситуаций. При использовании курсоров для этого потребовался новый и отличающийся режим блокировки. Вместо того, чтобы добавлять специальный код в менеджер блокировок, мы смогли создать альтернативную матрицу блокировок, в которой поддерживаются только режимы блокировок, необходимые для блокировки на уровне API. Таким образом, мы смогли получить необходимые нам режимы блокировок просто при помощи задания другой конфигурации менеджера блокировок. (К сожалению, изменить методы доступа оказалось не так легко; еще есть большие куски кода в методах доступа, в которых осуществляется обработка этого специального режима параллельного доступа).

4.8. Менеджер журнала: Log

Менеджер журнала предоставляет абстрактную модель структурированного файла, позволяющего только добавлять записи. Как и в других модулях, мы намеревались разработать универсальные средства записи в журнал, впрочем, подсистема ведения журнала, вероятно, является, как минимум, тем модулем, где мы добились успеха.

Одиннадцатый урок конструирования

Когда вы обнаруживаете проблемы в архитектуре и не хотите исправлять их «прямо сейчас», а склонны просто их пропустить, помните, что закусанные утками, вы умрете точно также, как если бы вас затоптали слоны. Для улучшения структуру программы без колебаний целиком меняйте целые фреймворки, а когда делаете изменения, не делайте частичные изменения в надежде на то, что позже вы приведете все в порядок — делайте все, а затем двигайтесь дальше. Часто повторяют: «если у вас нет времени сделать это прямо сейчас, у вас не найдется времени сделать это позже». И когда вы меняете фреймворки, пишите схемы тестирования.

Концептуально журнал сравнительно прост: он получает байтовые строки произвольной структуры и записывает их последовательно в файл, присваивая каждому уникальный идентификатор, называемый порядковым номером в журнале (log sequence number - LSN). Кроме того, журнал дол-

жен обеспечивать эффективный обход в прямом и обратном направлении и поиск по LSN. Есть два сложных момента: во-первых, журнал должен гарантировать, что он не будет испорчен после любого возможного отказа (что означает, что он содержит непрерывную последовательность неповрежденный журнальных записей), во-вторых, поскольку при подтверждении транзакций журнальные записи должны записываться в постоянное хранилище, производительность журнала является ,как правило, является именно тем, что ограничивает производительность любого приложения, использующего транзакции.

Поскольку журнал является структурой, в которую можно только добавлять записи, он может растя неограниченно. Мы реализуем журнал как совокупность последовательно пронумерованных файлов, так что место для журнала могут восстановить простым удалением старых журнальных файлов. Учитывая, что для журнала используется многофайловое решение, мы формируем номера LSN в виде пары, указывающей номер файла и смещение в файле. Таким образом, для заданного номера LSN журнальный менеджер тривиальным образом находит запись: он обращается по указанному смещению в заданном файле и возвращает запись, находящуюся в этом месте. Но как журнальный менеджер знает, сколько байтов вернуть из этого места?

4.8.1. Формат журнальных записей

Для того, чтобы для заданного номера LSN журнальный менеджер мог определить размер записи, которую он возвращает, в журнале с каждой записью должны храниться метаданные. Как минимум, нужно знать длину записи. Мы предваряем каждую журнальную запись заголовком записи, в котором указывается длина записи, смещение от предыдущей записи (для облегчения обратного обхода) и контрольная сумма журнальной записи (для идентификации разрушения журнала и определения конца журнального файла). Этих метаданных журнальному менеджеру достаточно для поддержки последовательно записываемых журнальных записей, но недостаточно для того, чтобы выполнить действительное восстановление; эти функциональные возможности закодированы в содержимом журнальных записей и определены тем, как Berkeley DB использует эти журнальные записи.

Berkeley DB использует журнальный менеджер для того, чтобы записывать образы данных перед обновлением элементов данных в базе данных и после их обновления [HR83]. В этих журнальных записях достаточно информации для того, чтобы либо повторить (действие redo), либо отменить (действие undo) операцию в базе данных. Berkeley DB использует журнал для отмены транзакции (то есть, отменяя все результаты выполнения транзакции при отмене транзакции) и для восстановления после сбоя приложения или системы.

В добавок, к интерфейсам API, предназначенным для чтения и сохранения журнальных записей, в журнальном менеджере предоставляется интерфейс API, позволяющий принудительно выгружать журнальные записи на диск (`DB_ENV->log_flush`). Это позволяет в Berkeley DB реализовать упреждающую запись в журнал – прежде, чем некоторая страница будет удалена из Mpool, Berkeley DB проверит номер LSN на странице и попросит журнальный менеджер обеспечить, чтобы указанный номер LSN был в постоянном хранилище. Только после этого Mpool записывает страницу на диск.

Двенадцатый урок конструирования

В Mpool и Log для того, чтобы упростить упреждающую запись в журнал, используются внутренние методы-обработчики и в некоторых ситуациях объявление метода по размеру больше, чем исполняемый код, поскольку код чаще всего лишь сравнивает два целых значения и больше ничего не делает. Зачем утруждать себя такими незначительными методами, только для возможности выделения слоев? Потому, что если ваш код не настолько объектно-ориентированный, чтобы вызывать зубную боль, то он не является достаточно объектно-ориентированным. В каждом куске кода нужно делать небольшую часть работы и должен быть более высокий уровень, на котором программистам будет предложено создавать функции из меньших фрагментов, и так далее. Если есть

что-нибудь, что мы узнали о разработке программ в последние несколько десятилетий, это то, что наши возможности создавать и поддерживать значительные фрагменты программ достаточно хрупки. Создание и поддержание значительной фрагментов программного обеспечения является сложным процессом, подверженным появлению ошибок, и, как архитектор программы, вы должны сделать все, что можно, так рано, настолько это можно, и делать настолько часто, насколько это возможно, чтобы максимизировать информацию, отображаемую в структуре вашей программы.

В Berkeley DB для того, чтобы облегчить восстановление, в журнальных записях задается определенная структура. Большинство журнальных записей Berkeley DB описывают транзакционные модификации. Таким образом, большинство журнальных записей отражают модификации страниц в базе данных, выполняемых в рамках транзакций. Исходя из этого можно это описание взять за основу для определения того, что должно указываться в метаданных Berkeley DB, добавляемых к каждой журнальной записи: база данных, транзакция и тип записи. Поля идентификатора транзакции и типа записи находятся в каждой записи на одном и том же месте. Это позволяет системе восстановления извлекать тип записи и перенаправлять запись в соответствующий обработчик, который может интерпретировать запись и выполнить соответствующие действия. Идентификатор транзакции позволяет процессу восстановления идентифицировать транзакцию, которой принадлежит журнальная запись, с тем чтобы на любой стадии восстановления было известно, можно ли эту запись проигнорировать или ее нужно обработать.

4.8.2. Разрушая абстракцию

Есть также несколько «специальных» журнальных записей. Среди таких специальных записей записи о контрольных точках являются, пожалуй, наиболее известными. Процесс создания контрольных точек является процессом запоминания на диске состояния базы данных в определенные моменты времени. Другими словами, Berkeley DB для улучшения производительности агрессивно кэширует в Mpool страницы баз данных. Но эти страницы должны быть, в конечном счете, записаны на диск, и чем скорее мы это сделаем, тем быстрее мы сможем сделать восстановление в случае отказа приложения или системы. При этом возможен компромисс между частотой создания контрольных точек и продолжительностью восстановления: чем чаще система создает контрольные точки, тем более быстро ее можно будет восстановить. Создание контрольных точек является транзакционной функцией, поэтому мы рассмотрим детали создания контрольных точек в следующем разделе. В данном разделе, в соответствие с его спецификой, мы рассмотрим записи о контрольных точках и о том, как журнальный менеджер выступает в двух ролях – как автономно работающий модуль и как компонент Berkeley DB специального назначения.

Обычно сам журнальный менеджер понятия не имеет о типах записей, так что теоретически он не должен отличать записи о контрольных точках от других записей – они просто являются строками байтов с произвольной структурой, которые журнальный менеджер записывает на диск. На практике, в журнале хранятся метаданные, а это указывает, что журнальный менеджер понимает содержание некоторых записей. Например, во время запуска журнала, журнальный менеджер проверяет все файлы, которые он может найти, и определяет, какой из них был записан последним. Он предполагает, что все журнальные файлы, записанные до этого, заполнены и он их не трогает, а начинает исследовать самый последний журнальный файл и пытается определить, сколько в нем содержится действительных журнальных записей. Он читает журнальный файл с начала и останавливается в случае, если / когда он обнаруживает заголовок журнальной записи, в котором находится неверная контрольная сумма, что указывает на конец журнала или начало испорченной части журнального файла. В любом случае это логический конец журнала.

В процессе этого чтения журнала, когда происходит поиск текущего конца журнала, журнальный менеджер извлекает тип записи Berkeley DB и ищет записи о контрольных точках. Он запоминает позицию последней найденной им записи о контрольной точке в метаданных журнального менеджера в качестве «предпочтительной» для системы транзакций. Т.е. последнюю контрольную точку должна искать система транзакций, но вместо того, чтобы позволить как журнальному менеджеру,

так и менеджеру транзакций обоим прочитывать весь журнальный файл, менеджер транзакций делегирует эту задачу журналному менеджеру. Это классический пример нарушения границ абстрагирования в обмен на повышение производительности.

Каковы последствия этого компромисса? Представьте, что система, отличная от Berkeley DB, использует журналный менеджер. Если случится так, что в той же самой позиции, где Berkeley DB размещает свой тип записи, будет записано значение, соответствующее типу записи о контрольной точке, то журналный менеджер идентифицирует эту запись, как запись о контрольной точке. Однако, если приложение запросит у журнального менеджера эту информацию (прямым доступом к полю `cached_ckp_1sn` в метаданных журнала), эта информация никак ни на что не влияет. Короче говоря, это либо вредное нарушение деления проекта на слои, либо хитроумная оптимизация производительности.

Управление файлами является еще одним местом, где разделение между журнальным менеджером и Berkeley DB является нечетким. Как уже упоминалось ранее, в большинстве журнальных записей Berkeley DB должна указываться база данных. В каждой журнальной записи должно быть полное имя файла базы данных, но это дорого с точки зрения расходования пространства журнала и громоздко, поскольку процесс восстановления должен использовать отображение этого имени в дескриптор определенного вида, который можно использовать для доступа к базе данных (дескриптор файла или дескриптор базы данных). Вместо этого, база данных в Berkeley DB идентифицируется в журнале по целочисленному идентификатору, называемому идентификатором файла и реализован набор функций, называемый `dbreg` (для «регистрации базы данных»), который поддерживает соответствие между именами файлов и идентификаторами журнальных файлов. Вариант такого отображения, хранящийся на диске (имеющий тип записи `DBREG_REGISTER`), заносится в журнальные записи при открытии базы данных. Тем не менее, для того, чтобы облегчить отмену транзакций и выполнения восстановления, нам также нужны варианты представления этого отображения в оперативной памяти. На какую из подсистем возложить обязанность поддержки этого отображения?

Теоретически, отображение файла в идентификатор журнального файла является высокоуровневой функцией Berkeley DB; она не принадлежит ни к одной из подсистем, и она не вписывается в общую картину. В исходном варианте проекта информация об отображении оставалась в структурах данных журнальных подсистем, посколькуказалось, что журнальная система является лучшим вариантом. Тем не менее, после повторного поиска и исправления ошибок в реализации, поддержка отображения была изъята из кода журнальной подсистемы и была преобразована в свою собственную небольшую подсистему со своим собственным объектно-ориентированным интерфейсом и собственными структурами данных. (В ретроспективе, эта информация должна логически размещаться в самой информационной среде Berkeley DB вне любой подсистемы).

Тринадцатый урок конструирования

Редко встречается такая вещь, как несущественная ошибка. Конечно, то и дело встречаются опечатки, но ошибка обычно означает, кто-то не понял в полной мере, что он должен сделать и что-то реализовал неправильно. Когда вы исправили ошибку, не ищите симптом: ищите причину, если хотите – непонимание, т.к. это приведет к лучшему пониманию структуры проекта, а также к выявлению фундаментальных недостатков в самом проекте.

4.9. Менеджер транзакций: Txn

Нашим последним модулем является менеджер транзакций, который связывает вместе отдельные компоненты и реализует транзакционные свойства ACID — атомарность (atomicity), целостность (consistency), изолированность (isolation) и долговечность (durability). Менеджер транзакций ответственен за начало и за завершение транзакций (транзакция либо выполняется, либо отменяется), за координацию с журналным менеджером и с менеджером буферирования при создании кон-

трольных точек транзакций и за общее управление процессом восстановления. Мы по порядку рассмотрим каждую из этих тем.

Джим Грей придумал сокращение ACID для описания ключевых свойств, предоставляемых транзакциями [Gra81]. Atomicity или атомарность означает, что все операции, выполняемые в рамках транзакции, осуществляются в базе данных как единое целое – либо они все выполняются в базе данных, либо они все не выполняются. Consistency или согласованность означает, что транзакция переводит базу данных из одного логически согласованного состояния в другое согласованное состояние. Например, если в приложении указывается, что все сотрудники должны быть назначены в отдел, который описан в базе данных, то это реализуется с помощью свойства согласованности (при правильно написанных транзакциях). Isolation или изолированность означает, что если рассматривать транзакцию, то она выполняется как бы последовательно, а не параллельно с какими-нибудь другими транзакциями. Наконец, durability или долговечность означает, что если транзакция выполнена, то она остается выполненной — никакой сбой не сможет отменить выполненную транзакцию.

Подсистема транзакций с помощью других систем осуществляет реализацию свойств ACID. В ней используются традиционные транзакционные операции `begin` (начало транзакции), `commit` (подтверждение выполнения транзакции) и `abort` (отмена выполнения транзакции), указывающие начальную и конечную точки транзакции. В ней также реализуется вызов вида `prepare call`, который помогает осуществлять двухфазное подтверждение выполнения транзакций — технологии реализации транзакционных свойств для распределенных транзакций, которые в данной главе не рассматриваются. Транзакционная операция `begin` создает идентификатор новой транзакции и возвращает в приложение дескриптор транзакции `DB_TXN`. Транзакционная операция `commit` создает журнальную запись `commit`, а затем инициирует запись журнала на диск (если в приложении не указано, что оно готово оказаться от свойства долговечности в обмен на более быструю обработку операции `commit`) , что гарантирует, что даже при наличии отказа транзакция будет выполнена. Транзакционная операция `abort` выполняет чтение в обратном направлении журнальных записей, принадлежащих указанной транзакции, отменяя каждую операцию, которая была сделана транзакцией и возвращает базу данных в состояние, предшествующее выполнению транзакции.

4.9.1. Обработка контрольных точек

На менеджер транзакций также возлагается задача создания контрольных точек. В литературе описан ряд различных методик создания контрольных точек [HR83]. В Berkeley DB используется вариант нечетких контрольных точек. По большому счету при создании контрольных точек необходимо записывать на диск состояние буферов в Mpool. Это потенциально дорогостоящая операция, и для того, чтобы избежать длительных отказов в обслуживании, важно, чтобы система, когда она это выполняет, продолжала обрабатывать новые транзакции. Перед тем, как создать контрольную точку, Berkeley DB сначала просматривает множество активных в данный момент транзакций и ищет наименьший номер LSN, записанный какой либо из этих транзакций. Этот номер LSN становится номером контрольной точки LSN. Затем менеджер транзакций просит Mpool сбросить на диск все буферы, в которых были сделаны изменения; запись этих буферов может, в свою очередь, вызвать выполнение операций записи на диск журнала. После того, как все буферы будут надежно сохранены на диске, менеджер транзакций затем записывает в журнал запись о контрольной точке, в которой указывается номер LSN. Эта запись указывает, что результаты всех операций, записанных в журнальных записях перед контрольной точкой с номером LSN, надежно сохранены на диске. Поэтому журнальные записи, находящиеся перед записью о контрольной точке с номером LSN, больше не требуются для восстановления. Это подразумевает следующее: во-первых, система может повторно использовать место в журнальных файлах, находящихся перед записью о контрольной точке LSN. Во-вторых, при восстановлении нужно обрабатывать только записи, находящиеся после контрольной точки LSN, поскольку обновления, описываемые в записях перед контрольной точкой LSN, уже отражены в состоянии на диске.

Обратите внимание, что между записью о контрольной точке с номером LSN и записью о фактической контрольной точке может быть много журнальных записей. Это нормально, поскольку в этих записях описываются операции, которые логически произошли после создания контрольной точки и в случае, если произойдет отказ системы, их, возможно, потребуется восстановить.

4.9.2. Восстановление

Последней частью транзакционной головоломки является процесс восстановления. Цель восстановления — перевести базу данных, хранящуюся на диске, из потенциально несогласованного состояния в согласованное состояние. В Berkeley DB используется довольно обычная двухпроходная схема, которая, в общих чертах, «связана с последней контрольной точкой с номером LSN, отменой всех транзакций, которые никогда не были подтверждены, и повторного выполнения транзакций, которые были подтверждены». Детали выглядят несколько сложнее.

Для того, чтобы Berkeley DB могла повторить или отменить операции, выполненные в базах данных, необходимо реконструировать отображение между идентификаторами журнальных файлов и фактическими базами данных. В журнале хранится вся история записей DBREG_REGISTER, но поскольку базы данных остаются открытыми в течение длительного времени и мы не хотим требовать, чтобы журнальные файлы не изменялись в течение того времени, пока база данных открыта, нам, вероятно, нужен более действенный способ доступа к этому отображению. Прежде, чем делать запись о контрольной точке, менеджер транзакций записывает серию записей DBREG_REGISTER, описывающих текущее отображение между идентификаторами журнальных файлов и базами данных. Во время восстановления Berkeley DB использует эти журнальные записи для восстановления отображения файлов.

Когда начинается процесс восстановления, менеджер транзакций для того, чтобы определить в журнале место последней записи о контрольной точке, берет из журнального менеджера значение cached_ckp_lsn. В этой записи есть номер контрольной точки LSN. Berkeley DB должна осуществлять восстановление от этой контрольной точки LSN, но для того, чтобы это сделать, она должна реконструировать отображение идентификаторов журнальных файлов, которое было в момент создания контрольной точки LSN; эта информация есть для контрольной точки, находящейся *перед* контрольной точкой LSN. Таким образом, Berkeley DB должна искать последнюю запись о контрольной точке, которая находится перед контрольной точкой с номером LSN. Чтобы облегчить этот процесс, в записях о контрольных точках есть не только номер контрольной точки LSN, но и номер LSN предыдущей контрольной точки. Процесс восстановления начинается с последней контрольной точки и происходит в обратном направлении по записям о контрольных точках до тех пор, пока не будет найдена запись о контрольной точке, находящаяся перед записью о контрольной точке с номером LSN. Переход от одной записи к другой происходит с использованием поля prev_lsn, которое есть в каждой записи о контрольной точке. Алгоритм следующий:

```
ckp_record = read (cached_ckp_lsn)
ckp_lsn = ckp_record.checkpoint_lsn
cur_lsn = ckp_record.my_lsn
while (cur_lsn > ckp_lsn) {
    ckp_record = read (ckp_record.prev_ckp)
    cur_lsn = ckp_record.my_lsn
}
```

Чтобы реконструировать отображения идентификаторов журнальных файлов, процесс восстановления выполняет последовательное чтение, которое начинается с контрольной точки, выбранной в предыдущем алгоритме, до конца журнала. Когда будет достигнут конец файла, восстановленные отображения будут точно соответствовать отображениям, которые были в момент остановки системы. Также во время этого прохода отслеживаются все встретившиеся записи с операцией подтверждения транзакций commit и записываются их идентификаторы транзакций. Любая транзакция, для которой есть журнальные записи, но идентификатор транзакций отсутствует в записи операции подтверждения commit, либо была отменена, либо никогда не завершалась и ее следует

трактовать как отмененную. Когда процесс восстановления достигнет конца журнала, направление движения изменится на обратное, и журнал будет читаться в обратном направлении. Для того, чтобы определить нужно ли для встретившейся журнальной записи о транзакции делать отмену операции, из каждой такой записи будет взят идентификатор транзакции, который будет сверен со списком транзакций, выполнение которых было подтверждено. Если процесс восстановления определит, что этот идентификатор транзакций отсутствует в списке подтвержденных транзакций, он определит тип записи и вызовет для этой журнальной записи процедуру восстановления, указывая ей отменить описанную операцию. Если этот идентификатор транзакций есть в списке подтвержденных транзакций, то процесс восстановления будет игнорировать ее на обратном проходе. Этот обратный проход будет продолжаться до контрольно точки с номером LSN [1]. Наконец, процесс восстановления прочитает журнал в последний раз в прямом направлении, на этот раз повторно выполняя действия для каждой журнальной записи, принадлежащей к подтвержденным транзакциям. Когда этот финальный проход будет завершен, процесс восстановления создаст контрольную точку. В этой точке база данных будет полностью согласованной и будет готова к началу работы приложения.

Таким образом, процесс восстановления можно резюмировать следующим образом:

1. Находим среди недавних контрольных точек контрольную точку, которая предшествует контрольной точке с номером LSN.
2. Читаем журнал в прямом направлении для того, чтобы восстановить отображения идентификаторов журнальных файлов, и создаем список завершенных транзакций.
3. Читаем обратно до контрольной точки с номером LSN, отменяя все операции для незавершенных транзакций.
4. Читаем в прямом направлении, повторно выполняя все операции для подтвержденных транзакций.
5. Создаем контрольную точку.

Теоретически заключительная контрольная точка не нужна. На практике она ограничивает время при будущих восстановлениях и позволяет оставаться базе данных в согласованном состоянии.

Четырнадцатый урок конструирования

Процесс восстановление базы данных является сложной темой, его трудно написать и еще труднее отладить, поскольку он не должен происходить часто. В своей лекции при получении премии Тьюринга, Эдсгер Дейкстра (Edsger Dijkstra) утверждал, что программирование является трудным по своей сути и нужно признать, что мы не в состоянии справиться с этой задачей. Наша цель, как архитекторов и программистов, заключается в использовании имеющихся в нашем распоряжении инструментов: проектирования, декомпозиции проблем, просмотров, тестирования, конвенций об именах и стилях и других хороших приемов, сводящих проблемы программирования к задачам, которые мы можем решить.

4.10. Заключение

В настоящее время проекту Berkeley DB более двадцати лет. Это, возможно, было первое транзакционное хранилище типа "ключ/значение" и он является прародителем движения NoSQL. Система Berkeley DB продолжает использоваться в качестве основной системы хранения в сотнях коммерческих продуктов и тысячах приложений с открытым кодом (включая движки SQL, XML и NoSQL) и установлена на миллионах компьютерах по всему миру. Уроки, которые мы получили в ходе ее развития и сопровождения, отражены в ее коде и обобщены в советах по конструированию, изложенных выше. Мы предлагаем их в надежде, что другие разработчики программного обеспечения и архитекторы посчитают их полезными.

Примечания

1. Обратите внимание, что нам нужен обратный проход только до контрольной точки с номером LSN, а не до контрольной точки, предшествующей ей.

5. СMake

В 1999 году Национальная медицинская библиотека заказала небольшой компании, называющейся Kitware, разработать улучшенный способ конфигурирования, сборки и развертывания сложного программного обеспечения, предназначенный для большого количества различных платформ. Эта работа была частью проекта Insight Segmentation and Registration, или ITK (смотрите в конце статьи примечание 1). На компанию Kitware, играющую ведущую роль в проекте, было возложена разработка системы сборки приложений, которой могли бы пользоваться исследователи и разработчики проекта ITK. Система должна была быть простой в использовании и должна была помочь наиболее продуктивно использовать время, затрачиваемое исследователями на программирование. Согласно этой директиве появилась система CMake, которая заменила устаревающий подход с использованием autoconf/libtool, применявшийся при сборке программ. Система была разработана для того, чтобы преодолеть недостатки существующих инструментальных средств и при этом сохранить их преимущества.

Кроме того, что CMake является системой сборки, он в течение многих лет эволюционировал в семейство инструментальных средств: CMake, CTest, CPack и CDash. CMake является инструментом для сборки, предназначенным для создания программ. CTest - это инструментальный драйвер тестов, применяемый при запуске регрессионных тестов. CPack является упаковщиком, используемым при создании инсталляторов на конкретные платформы для программ, созданных с использованием CMake. CDash - это веб-приложение, предназначенное для отображения результатов тестирования и выполнения тестирования в технологии непрерывной сборки проектов.

5.1. История создания CMake и предъявленные к нему требования

Когда разрабатывался CMake, обычной практикой было использование в проекте конфигурационного скрипта и файлов Makefile для платформ Unix, а для Windows - проектных файлов Visual Studio. Такая двойственность систем сборки делала кроссплатформенную разработку во многих проектах очень утомительной: простое действие по добавлению в проект нового файла с исходным кодом оказывалось тяжелым делом. Очевидно, что разработчики хотели иметь единую унифицированную систему сборки. У разработчиков CMake был опыт двух подходов к решению проблемы унифицированной системы сборки.

Первым подходом была система сборки [VTK](#) разработки 1999 года. Эта система состояла из конфигурационного скрипта для Unix и исполняемого модуля для Windows, называемого `rcmaker`. Программа `rcmaker`, которая была написана на языке C, читала файлы Makefile для платформ Unix и создавала файлы NMake - для Windows. Исполняемый модуль `rcmaker` помещался в репозиторий CVS системы VTK. В некоторых типичных случаях, например, при добавлении новой библиотеки, нужно было изменять исходный код этого модуля и новый двоичный модуль снова помещать в репозиторий. Хотя, в некотором смысле, это была унифицированная система, в ней было много недостатков.

Другой подход, опыт применения которого был у разработчиков, состоял в использовании системы `gmake`, представляющей собой базовую систему сборки для системы TargetJr. Система TargetJr являлась средой машинного зрения, написанной на языке C++ и первоначально разработанной для рабочих станций фирмы Sun. Сначала в TargetJr при создании файлов Makefile использовалась система `imake`. Но, в какой-то момент, когда потребовался порт для Windows, была создана система `gmake`. С этой системой, базирующейся на `gmake`, можно было использовать как компиляторы для платформы Unix, так и компиляторы для Windows. Системе требовалось несколько перемен-

ных среды окружения, которые нужно было установить перед запуском gmake. Неверная настройка среды окружения приводила к сбою в работе системы, причем такому, в котором было трудно разобраться, особенно - конечным пользователям.

Оба эти подхода страдали от серьезного недостатка: они заставляли разработчиков приложений для Windows использовать командную строку. Опытные разработчики предпочитают в Windows использовать интегрированные среды разработки (IDE). Это побуждало разработчиков приложений для Windows вручную создавать файлы IDE и добавлять их в проект, снова создавая двойную систему сборки. Кроме отсутствия поддержки среды IDE, в обоих подходах, описанных выше, было чрезвычайно трудно совмещать вместе различные программные проекты. Например, в VTK было очень мало модулей чтения изображений, главным образом, из-за того, что в системе сборки было очень трудно пользоваться библиотеками, например, libtiff и libjpeg.

Было решено, что новая система сборки должна быть разработана для ITK и, в общем случае, для C++. Основные требования к новой системе сборки выглядели следующим образом:

- Зависимость должна быть только от компилятора C++, установленного в системе.
- Должна быть возможность генерировать входные файлы для Visual Studio IDE.
- Должна быть возможность указывать системе сборки, что надо собрать - статические библиотеки, разделяемые библиотеки, исполняемые модули или плагины.
- Должна быть возможность на этапе сборки запускать генераторы кода.
- Поддержка деревьев сборки должна быть отделена от поддержки дерева исходных кодов.
- Должна быть возможность выполнять самопроверку системы, то есть возможность автоматически определять, что сможет и что не сможет делать система.
- Должно автоматически выполняться сканирование зависимостей по заголовочным файлам C/C++.
- Все функции должны работать слаженно и одинаково на всех поддерживающих платформах.

Чтобы избежать зависимости от каких-либо дополнительных библиотек и анализаторов, CMake был спроектирован только с одной главной зависимостью — компилятором C++ (который, как мы можем с уверенностью предположить, у нас есть, если мы собираем код C++). В то время было трудно собирать и устанавливать скриптовые языки, например, Tcl, на многих популярных системах UNIX и Windows. Сегодня на современных суперкомпьютерах и защищенных компьютерах, у которых нет подключения к интернету, это все еще остается проблемой, поскольку все еще трудно собирать библиотеки сторонних разработчиков. Т.к. система сборки, как таковая, являлась главной задачей, было решено, что в CMake не будут добавляться какие-либо дополнительные зависимости. Из-за этих ограничений в CMake пришлось создать свой собственный простой язык, из-за которого некоторым все еще не нравится CMake. Впрочем, тогда самым популярным встроенным языком был Tcl. Если бы CMake был системой сборки на основе Tcl, то вряд ли бы он достиг той популярности, которой он обладает сегодня.

Возможность создавать проектные файлы IDE является сильной стороной CMake, но это также является ограничением CMake, поскольку поддерживаются только те возможности, которые изначально присутствуют в IDE. Впрочем, преимущества получения сборочных файлов для конкретной IDE перевешивают эти ограничения. Хотя такое решение сделало разработку CMake сложнее, оно существенно облегчило разработку ITK и других проектов, в которых применяется CMake. Разработчикам удобнее и продуктивнее работать с теми инструментальными средствами, с которыми они лучше знакомы. Благодаря тому, что разработчики пользуются инструментальными средствами, которым они отдают предпочтение, в проектах можно наилучшим образом использовать самый важный ресурс - разработчиков.

Всем программам C/C++ необходим один или несколько следующих сборочных блоков: исполняемые модули, статические библиотеки, разделяемые библиотеки и плагины. CMake должен был позволять создавать эти компоненты для всех поддерживаемых платформ. Несмотря на то, что

создание таких компонентов поддерживается на всех платформах, флаги компилятора, которые используются при их создании, существенно различаются в зависимости от компиляторов и платформ. За счет того, что в CMake сложность и различие платформ скрыты в простой команде, разработчики могут создавать такие компоненты в Windows, Unix и Mac. Это позволяет разработчикам сосредоточиться на проекте, а не на деталях, связанных со сборкой разделяемой библиотеки.

В системе сборки дополнительная сложность связана с генераторами кода. В VTK с самого начала была предложена схема, в которой код на C++ автоматически помещается внутрь кода Tcl, Python и Java следующим образом: выполняется анализ заголовочных файлов C++ и автоматически создается слой-обертка. Для этого необходима система сборки, которая может собрать исполняемый модуль на C/C++ (генератор обвертки), а затем на этапе сборки запустить этот модуль и создать другой исходный код C/C++ (обертки для конкретных модулей). Затем этот сгенерированный исходный код должен быть откомпилирован в исполняемые модули или разделяемые библиотеки. Все это должно произойти в среде IDE и с использованием сгенерированных файлов Makefile.

Когда на C/C++ разрабатываются универсальные кроссплатформенные программы, важно программировать функциональные возможности системы, а не конкретную систему. В autotools есть модель для проведения самопроверки системы, которая состоит из компиляции небольших фрагментов кода, его проверки и запоминания результатов. Поскольку подразумевалось, что CMake должен был быть кроссплатформенным, в него вошла подобная технология самопроверки системы. Она позволяет разработчикам выполнять программирование для канонической системы, а не для конкретных систем. Это важно для достижения мобильности в будущем, поскольку со временем меняются как компиляторы, так и операционные системы. Например, следующий код:

```
#ifdef linux
// делаем что-то для linux
#endif
```

менее устойчив, чем следующий код:

```
#ifdef ИМЕЕТСЯ_ФУНКЦИЯ
// делаем что-то с помощью этой функции
#endif
```

Другое первоначальное требование к CMake также было взято из autotools: возможность создавать деревья сборки, которые будут отделены от дерева исходных кодов. Это позволяет для одного и того же дерева исходных кодов создавать несколько типов деревьев сборки. В результате предотвращается влияние деревьев сборки на дерево исходных кодов, что часто мешает работе систем контроля версий.

Одной из наиболее важных особенностей системы сборки является возможность управлять зависимостями. Если исходный файл изменен, то все программы, использующие этот исходный файл, должны быть пересобраны заново. Для кода C/C++ как часть зависимостей должны быть также проверены заголовочные файлы, включенные в файл .c или .cpp. Если не отслеживать случаи, когда в действительности должна компилироваться только часть кода, то из-за неверной информации о зависимостях может затрачиваться большое количество времени.

Все требования к новой системе сборки и ее функциональные возможности должны быть одинаково хорошо реализованы для всех поддерживаемых платформ. Необходимо, чтобы CMake был простым API, позволяющим разработчикам, не знающим особенностей платформы, создавать сложные программные системы. В сущности, программы, использующие CMake, перенаправляют в CMake всю сложность, связанную со сборкой. После того, как было создано общее представление об инструментальном средстве сборки и базовом наборе к нему требований, потребовалось достаточно гибко выполнить реализацию. В проекте ITK система сборки нужна была практически с первого дня. Первые версии CMake не отвечали всему набору требований это общего представления, но могли выполнять сборку на Windows и Unix.

5.2. Как реализован CMake

Как уже упоминалось, языками разработки пакета CMake являются С и С++. Чтобы объяснить его внутренние особенности, далее в настоящем разделе сначала описывается процедура использования CMake в том виде, как ее видит пользователь, а затем изучаются используемые в CMake структуры.

5.2.1. Процедура использования CMake

Процедура использования CMake состоит из двух основных этапов. Во-первых, это шаг "конфигурирования", на котором пакет CMake обрабатывает все переданные ему входные данные и создает внутреннее представление сборки, которую нужно выполнить. Затем идет следующий этап - шаг "генерации". На этом этапе создаются файлы фактической сборки.

Переменные среды окружения (или не среды окружения)

Во многих системах сборки в 1999 году и даже сегодня во время создания проекта используются переменные среды окружения командной оболочки. Закономерно, что в проекте есть переменная среды окружения PROJECT_ROOT, которая указывает на местонахождение корня дерева исходных кодов. Переменные среды окружения также используются для указания на дополнительные или внешние пакеты. Трудность этого этого подхода в том, что для того, чтобы сборка проекта работала, нужно каждый раз, когда выполняется сборка, устанавливать значения всех этих внешних переменных. Чтобы решить эту проблему, в CMake есть кэш-файл, в котором в одном месте запомнены все переменные, необходимые для сборки. Они являются переменными CMake, а не переменными командной оболочки или среды окружения. Когда CMake запускается первый раз для конкретного дерева сборки, он создает файл CMakeCache.txt, в котором постоянно хранятся все переменные, необходимые для этой сборки. Поскольку этот файл является частью дерева сборки, переменные всегда будут доступны для CMake при каждом его запуске.

Шаг конфигурирования

На шаге конфигурирования CMake сначала читает файл CMakeCache.txt, если тот существует после предыдущего запуска. Затем он читает файл CMakeLists.txt, который находится в корне дерева исходных кодов, переданных в CMake. На шаге конфигурирования файл CMakeLists.txt анализируется анализатором языка CMake. Каждая команда CMake, обнаруженная в файле, выполняется объектом с образом команды. Кроме того, на этом шаге с помощью команд CMake include и add_subdirectory может быть проанализирован файл CMakeLists.txt. В CMake для каждой команды, которая может использоваться в языке CMake, есть объект C++. Примерами таких команд являются команды add_library, if, add_executable, add_subdirectory и include. В сущности, весь язык CMake реализован в виде обращений к командам. Синтаксический анализатор просто преобразует входные файлы CMake в вызовы команд и списки строк, которые являются аргументами этих команд.

На шаге конфигурирования, по существу, "работает" код CMake, предоставленный пользователями. После того, как весь код будет выполнен и будут вычислены все значения кэш-переменных, в CMake будет запомненное представление проекта, который должен быть собран. В него будут включены все библиотеки, исполняемые модули, команды пользователя и вся другая информация, необходимая для создания окончательных файлов сборки для выбранного генератора. В этом момент файл CMakeCache.txt сохраняется на диске для использования при будущих запусках CMake.

Запомненное представление проекта является набором целевых задач, определяющих просто то, что нужно собрать, например, библиотеки и исполняемые модули. В CMake также можно указывать пользовательские целевые задачи: пользователи могут определять свои собственные входные

и выходные данные, а также предоставлять свои собственные исполняемые модули или скрипты, которые должны работать во время сборки. В CMake каждая целевая задача хранится в объекте `cmTarget`. Эти объекты хранятся, в свою очередь, в объекте `cmMakefile`, который, по существу, является местом хранения всех целевых задач, найденных в заданном каталоге дерева исходного кода. Итоговым результатом является дерево объектов `cmMakefile`, содержащих отображения объектов `cmTarget`.

Шаг генерации

Как только шаг конфигурирования будет завершен, происходит переход к шагу генерации. Шаг генерации — это этап, когда CMake создает файлы сборки для целевого инструментального средства сборки, выбранного пользователем. В этот момент внутреннее представление целевых задач (создание библиотек, исполняемых модулей, пользовательские целевые задачи) преобразуется либо во входные файлы сборочной инструментальной среды IDE, такой как Visual Studio, либо в набор файлов Makefile, который будут обработан командой `make`. Внутреннее представление, создаваемое CMake после шага конфигурирования, настолько обобщено, что один и тот же код и те же самые структуры данных могут максимальным образом использоваться различными инструментальными средствами сборки.

Общий вид процедуры использования CMake приведен на рис.5.1.

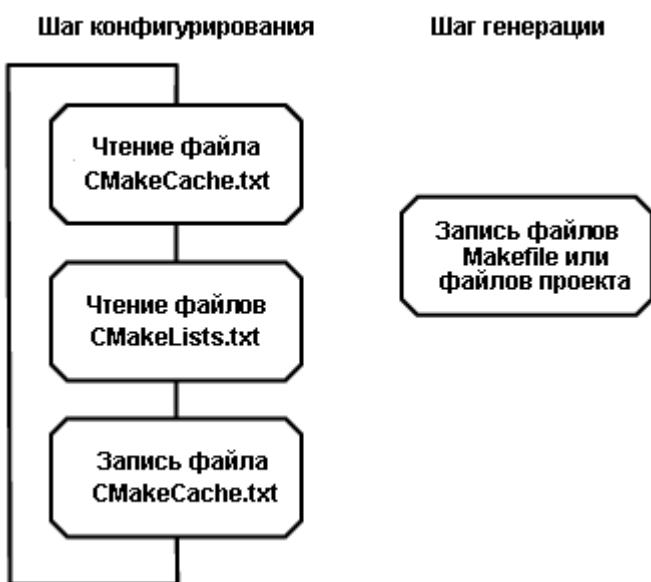


Рис.5.1: Общий вид процедуры использования CMake

5.2.2. CMake: Код

Объекты CMake

CMake является объектно-ориентированной системой, в которой используется наследование, шаблоны проектирования и инкапсуляция. Основные объекты на C++ и их отношения приведены на рис.5.2.



Рис.5.2: Объекты CMake

Результаты анализа каждого файла CMakeLists.txt хранятся в объекте cmMakefile. Объект cmMakefile не только хранит информацию о каталоге, но и управляет анализом файла CMakeLists.txt. Функция анализа обращается к объекту, который для анализа языка CMake обращается к синтаксическому анализатору на базе lex/yacc. Т.к. синтаксис языка CMake меняется не часто, а lex и yacc не всегда есть в системах, где CMake выполняет сборку, файлы, получаемые на выходе lex и yacc, обрабатываются и вместе со всеми другими файлами, созданными вручную, сохраняются в каталоге Source системы контроля версий.

Другим важным классом в CMake является класс cmCommand. Это базовый класс реализации всех команд языка CMake. В каждом его подклассе не только реализуется команда, но и осуществляется ее документирование. В качестве примера взгляните на методы документирования в классе cmUnsetCommand:

```

virtual const char* GetTerseDocumentation()
{
    return "Unset a variable, cache variable, or environment variable.";
}

/**
 * More documentation.
 */

virtual const char* GetFullDocumentation()
{
    return
        " unset( [CACHE])\n"
        "Removes the specified variable causing it to become undefined.  "
        "If CACHE is present then the variable is removed from the cache "
        "instead of the current scope.\n"
        " can be an environment variable such as:\n"
        "  unset(ENV{LD_LIBRARY_PATH})\n"
        "in which case the variable will be removed from the current "
        "environment.";
}
  
```

Анализ зависимостей

В CMake есть мощные встроенные средства анализа зависимостей для отдельных файлов исходного кода на языках Fortran, С и С++. Поскольку в интегрированных средах разработки (в IDE) есть свои собственные средства работы с информацией о зависимостях файлов, CMake для этих систем сборок пропускает шаг анализа зависимостей. В случае с IDE CMake создает входной файл, нативный для IDE, и предоставляет самой IDE обрабатывать информацию о зависимостях уровня файлов. Информация о зависимостях уровня целевой задачи преобразуется в формат IDE и добавляется к остальной информации о зависимостях.

Что касается сборок, использующих файлы Makefile, то до сих пор нативная программа make не знает, как автоматически вычислять и сохранять информацию о зависимостях. Для этих сборок CMake автоматически вычисляет информацию о зависимостях для файлов на С, С++ и Fortran. Вычисление информации о зависимостях и поддержание ее в актуальном состоянии автоматически выполняется с помощью CMake. После того, как с помощью CMake проект будет первоначально сконфигурирован, пользователям нужно будет только запускать команду `make`, а остальную работу сделает CMake.

Хотя пользователям не требуется знать, как CMake выполняет эту работу, может оказаться полезным взглянуть в проекте на файлы с информацией о зависимостях. Для каждой целевой задачи эта информация хранится в следующих четырех файлах - `depend.make`, `flags.make`, `build.make` и `DependInfo.cmake`. В файле `depend.make` хранится информация о зависимостях для всех объектных файлов каталога. В файле `flags.make` хранятся флаги компиляции, используемые с исходными файлами для этой целевой задачи. Если они изменились, то файлы должны быть перекомпилированы. Файл `DependInfo.cmake` все еще используется для хранения информации о зависимостях и также содержит информацию о том, какие файлы являются частями проекта и какие в них используются языки. Наконец, правила для сборки зависимостей хранятся в файле `build.make`. Если зависимости для целевой задачи устарели, информация о зависимостях для данной целевой задачи будет пересчитана заново и зависимости будут обновлены. Это делается из-за того, что любое изменение в любом файле .h может добавить новую зависимость.

CTest и CPack

По ходу дела CMake перерос из системы сборки проекта в семейство инструментальных средств, предназначенных для сборки, тестирования и создания пакетов программ. Кроме команды `cmake`, работающей из командной строки, и программ графического интерфейса CMake, CMake также поставляется в комплекте с инструментальным средством тестирования CTest и средством создания пакетов CPack. CTest и CPack используют тот же самый базовый код, что и CMake, но являются отдельными инструментами и для основной сборки проекта они не нужны.

Модуль `ctest` предназначен для запуска регрессионных тестов. В проекте можно легко с помощью команды `add_test` создавать тесты для CTest. Тесты можно выполнить с помощью CTest, а результаты тестирования также можно с помощью CTest отправить в приложение CDash для просмотра их в интернете. CTest и CDash вместе аналогичны инструментальному средству тестирования Hudson. Самое главное их отличие в следующем: CTest позволяет выполнять тестирование в более разнообразных условиях. Клиентские системы можно настроить таким образом, что исходный код будет выбираться из системы контроля версий, затем будут выполняться тесты, а результат будет отправляться в приложение CDash. В случае с Hudson для того, чтобы можно было запускать тесты, на клиентских машинах нужно открывать доступ пакету Hudson по ssh.

Модуль `cpack` предназначен для создания инсталляторов проектов. Пакет CPack работает почти также, как и та часть CMake, которая выполняет сборку: пакет взаимодействует с другими инструментальными средствами создания пакетов. Например, в Windows для создания в проекте самостоятельно запускаемых инсталляторов используется упаковщик NSIS. CPack запускает на выполнение инсталляционные правила проекта, в результате чего создается инсталляционное дерево, которое затем передается программе, создающей инсталляторы, например, NSIS. CPack также

поддерживает создание файлов RPM, файлов .deb — для Debian, а также файлов .tar, .tar.gz и самораспаковывающихся файлов tar.

5.2.3. Графические интерфейсы

Первое, с чем сталкиваются многие пользователи, только что увидевшие CMake, это одна из программ пользовательского интерфейса CMake. В CMake есть два основных приложения пользовательского интерфейса: оконное приложение на базе Qt, а также приложение с текстовым интерфейсом, похожим на графический, работающее из командной строки. Эти графические интерфейсы используются в качестве графического редактора файла `CMakeCache.txt`. Это сравнительно простые интерфейсы с двумя кнопками - конфигурирование и генерация, используемыми для запуска основных этапов процесса CMake. Текстовый интерфейс доступен на Unix-платформах типа TTY и в Cygwin. Графический интерфейс Qt доступен на всех платформах. Примеры графического интерфейса приведены на рис. 5.3 и 5.4.

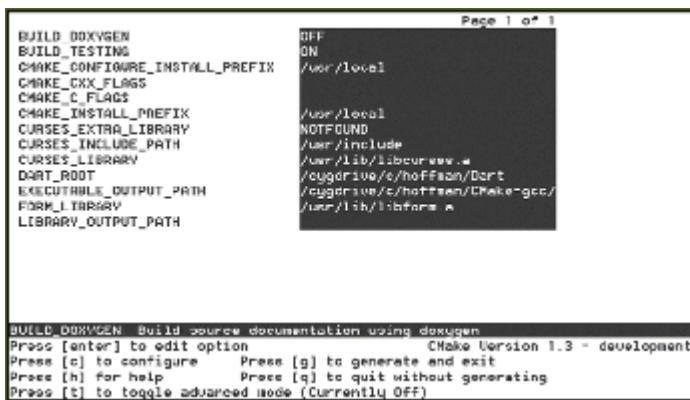


Рис.5.3: Интерфейс командной строки

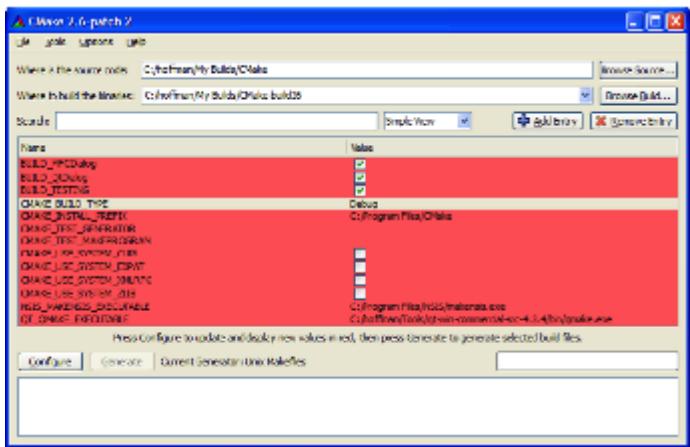


Рис.5.4: Графический интерфейс

В обоих вариантах графического интерфейса имена кэш-переменных размещены слева, а их значения - справа. Значения, указываемые справа, пользователь может заменить теми, которые подходят для конкретной сборки. Есть два набора переменных — обычный и расширенный. По умолчанию пользователю показывается обычный набор переменных. В проекте в файлах CMakeLists.txt можно указать, какие переменные будут входить в расширенный набор переменных. Этот подход позволяет предложить пользователю несколько вариантов, выбираемых для конкретной сборки.

Поскольку по мере того, как выполняются команды, значения кэш-переменных могут изменяться, процесс приближения к финальной сборке может быть итеративным. Например, включение некоторых параметров может потребовать использовать другие параметры. По этой причине в графи-

ческом интерфейсе кнопка "генерации" остается отключенной до тех пор, пока у пользователя не будет иметься возможности, по крайней мере, один раз увидеть все параметры. Каждый раз, когда нажимается кнопка конфигурирования, новые кэш-переменные, которые еще не были показаны пользователю, отображаются красным цветом. После того, как во время конфигурирования будут созданы все новые кэш-переменные, будет включена кнопка генерации.

5.2.4. Тестирование CMake

Любой новый разработчик CMake сначала знакомится с процессом тестирования, используемым при разработке CMake. В этом процессе задействовано семейство инструментальных средств CMake (CMake, CTest, CPack и CDash). Как только код разработан и помещен в систему контроля версий, машины, выполняющие тестирование в процессе непрерывной сборки проекта, автоматически выполняют сборку и с помощью пакета CTest тестируют новый код CMake. Результаты отсылаются на сервер CDash, который в случае, если есть ошибки сборки, предупреждения компилятора или если тест прошел неудачно, уведомляет об этом разработчиков по электронной почте.

Это классическая схема тестирования, используемая при непрерывной сборке проекта. Как только новый код помещается в репозиторий CMake, он автоматически тестируется на платформах, поддерживаемых в CMake. Если учесть огромное количество компиляторов и платформ, поддерживаемых в CMake, то такая схема тестирования является весьма важной для разработки стабильной системы сборки.

Например, если новый разработчик захочет добавить поддержку новой платформы, то первый вопрос, который ему будет задан — может ли обеспечить непрерывную работу клиентской программы CMake для этой системы. Без постоянного тестирования новая система через некоторое время неизбежно перестанет работать.

5.3. Усвоенные уроки

CMake с самого первого дня успешно использовался при сборке проекта ITK, и это было самой важной частью проекта. Если бы мы могли снова повторить разработку CMake, то мало что следовало бы изменить. Тем не менее, всегда есть то, что можно было бы сделать лучше.

5.3.1. Обратная совместимость

Поддержка обратной совместимости очень важна для команды разработчиков CMake. Основной целью проекта является сделать проще сборку программ. Когда команда проекта или разработчик выбирают CMake в качестве инструмента сборки, важно уважать этот выбор и нужно очень постараться, чтобы сборку можно было бы продолжать выполнять с помощью последующих релизов CMake. В CMake 2.6 реализована политика, что в случае, если предполагается, что некоторое поведение системы устарело, об этом следует выдавать предупреждение, но в системе само такое устаревшее поведение должно поддерживаться. В каждом файле CMakeLists.txt следует указать, какую предполагается использовать версию CMake. Новые версии CMake могут выдавать предупреждения, но они все равно должны собирать проект так, как это делалось в старых версиях.

5.3.2. Язык, язык, язык

Предполагалось, что язык CMake должен быть очень простым. Тем не менее, он является одним из основных препятствий, когда в новом проекте принимается решение об использовании CMake. С языком CMake в процессе его естественного роста были несколько интересных моментов. Первый синтаксический анализатор языка был создан даже не на базе lex/yacc, а был простым анализатором строк. Когда появился шанс расширить язык, мы потратили некоторое время на поиск хорошего встроенного языка среди уже имеющихся языков. Лучшим вариантом для работы оказался

язык Lua. Это очень маленький и ясный язык. Даже если бы не использовался сторонний язык, похожий на Lua, мне надо было с самого начала больше внимания уделять существующему языку.

5.3.3. Плагины не работают

Чтобы в проектах можно было расширять возможности языка CMake, в CMake есть класс плагинов. Он позволяет в проекте на языке C создавать новые команды CMake. На тот момент такой подход казался правильным и для того, чтобы можно было пользоваться другими компиляторами, был определен интерфейс языка C. Но с появлением большого количества систем API, например, для 32/64-разрядных Windows и Linux, совместимость плагинов стало поддерживать трудно. Несмотря на то, что расширение языка CMake с помощью самого CMake является не столь мощным средством, оно позволяет избежать ситуаций, когда CMake перестает работать или не может собрать проект из-за того, что плагин не удалось собрать или загрузить.

5.3.4. Ограничение распространения API

Важный урок, усвоенный в процессе разработки проекта CMake, состоит в том, что вы не должны поддерживать обратную совместимость с тем, к чему пользователи не должны получать доступ. Несколько раз в ходе разработки CMake пользователи и заказчики просили, чтобы CMake был преобразован в библиотеку для того, чтобы функциональные возможности CMake можно было бы добавить к другим языкам. Это бы не только разрушило сообщество пользователей CMake из-за того, что возникло бы множество способов применения CMake, но также чрезмерно увеличило бы затраты на техническое обслуживание проекта CMake.

Примечания

1. <http://www.itk.org/>

6. Среда разработки Eclipse

Глава 6 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Реализация модульности в программах является крайне трудной задачей. Также трудно управлять взаимодействием с большой базой кода, написанного различными представителями сообщества. В проекте Eclipse нам удалось добиться успеха в обоих случаях. В июне 2010 года фонд Eclipse Foundation предоставил свой релиз Helios, скоординированный с более чем 39 проектами и 490 участниками из более чем 40 компаний, которые работают совместно над разработкой функциональных возможностей базовой платформы. Каков было изначальное архитектурное видение проекта Eclipse? Как он развивался? Как архитектура приложения поощряет участие и повышает роль сообщества в его разработке? Давайте обратимся к истокам.

7 ноября 2001 года была выпущена версия 1.0 проекта с открытым исходным с названием Eclipse. В то время Eclipse описывался как «интегрированная среда разработки (IDE) для всего и для ничего конкретного, в частности». Такое описание было намеренно общим, поскольку с архитектурной точки зрения, это был не просто еще один набором инструментов, это был фреймворк, который был модульным и масштабируемым. Eclipse предоставил платформу, базирующуюся на компонентах, которая могла бы послужить основой для создания инструментария для разработчиков. В этой расширяемой архитектуре сообществу предлагалось опереться на основную платформу и расширять ее за пределы первоначальной концепции. Eclipse стартовал в качестве платформы, а Eclipse SDK был продуктом, предназначенным для проверки концепции. Eclipse SDK был предоставлен разработчикам в их личное распоряжение и одновременно использовался для создания новых версий Eclipse,

Стереотипный образ разработчика открытого исходного кода представляет собой альтруистическую личность, трудящуюся поздней ночью, исправляющую ошибки и реализующую новые фантастические возможности для решения своих частных интересов. Напротив, если вы взгляните на начало истории Eclipse, вы увидите, что часть исходного кода, который был подарен проекту, опирался на проект VisualAge for Java, разработанный IBM. Первые, кто сделал вклад в этот проект с открытым кодом, были сотрудники вспомогательного подразделения IBM, называющегося Object Technology International (OTI). Эти разработчики работали над проектом с открытым исходным кодом полный оплачиваемый рабочий день и отвечали на вопросы из групп новостей, находили ошибки и реализовывали новые возможности. Чтобы расширить эти усилия по созданию подобного открытого инструментария, был сформирован консорциум заинтересованных поставщиков программ. Первоначально членами консорциума Eclipse были Borland, IBM, Merant, QNX Software Systems, Rational Software, RedHat, SuSE и TogetherSoft.

Благодаря инвестициям своих усилий, эти компании получили опыт поставки коммерческих изделий, создаваемых на основе Eclipse. Это похоже на те инвестиции, которые корпорации вкладывают разработку ядра Linux, т. к. в собственных интересах компаний, чтобы сотрудники улучшали программное обеспечение с открытым исходным кодом, лежащем в основе их коммерческих предложений. В начале 2004 года был сформирован фонд Eclipse Foundation, который предназначался для управления и расширения растущего сообщества Eclipse. Этот некоммерческий фонд был профинансирован его корпоративными представителями и управление им осуществлялось советом директоров. Сегодня сообщество Eclipse расширилось еще сильнее и теперь включает более 170 компаний и почти 1000 крупных участников.

Поначалу Eclipse рассматривался только как SDK, но сегодня он представляет из себя намного большее. В июле 2010 года в eclipse.org в стадии разработки было 250 различных проектов. Среди них инструментарий для разработки на языках C/C++ и PHP, для разработки веб-сервисов, разработки на основе моделей, создания инструментальных средств и многое другое. Каждый из этих проектов включен в список проектов верхнего уровня (TLP), которые находятся в ведении Комитета управления проектами (PMC), состоящего из старших членов проекта, взявших на себя ответственность по поддержке технического направления и достижения конкретных целей. Для краткости в этой главе будут рассмотрены только проекты Eclipse SDK из Eclipse [1] и Runtime Equinox [2]. Т.к. Eclipse имеет долгую историю, я сосредоточусь на раннем варианте Eclipse, а также на версиях 3.0, 3.4 и 4.0.

6.1. Ранний вариант Eclipse

В начале 21-го века разработчикам программного обеспечения было предложено много инструментальных средств, но немногими из них можно было пользоваться одновременно. В проекте Eclipse старались создать платформу с открытым кодом, предназначенную для построения совместно работающих инструментальных средств для разработчиков приложений. Это должно было позволить разработчикам сосредоточиться на написании новых инструментов, а не писать код, который будет связан с вопросами инфраструктуры, например, с взаимодействием с файловой системой, работой с обновлениями и с подключением к хранилищам исходного кода. В проекте Eclipse, пожалуй, самым известным инструментальным средством, предназначенным для разработки на языке java, является Java Development Tools (JDT). Предполагалось, что это инструментальное средство для Java будет служить примером для тех, кто заинтересован в разработке инструментария для других языков.

Прежде, чем мы углубимся в архитектуру Eclipse, давайте посмотрим на то, как Eclipse SDK выглядит для разработчика. После запуска Eclipse и выбора рабочего пространства (workbench), вам будет представлен набор панелей для работы с Java (Java perspective). Такой набор состоит из специальных инструментальных панелей и панелей редакторов, которые необходимы в конкретном случае.

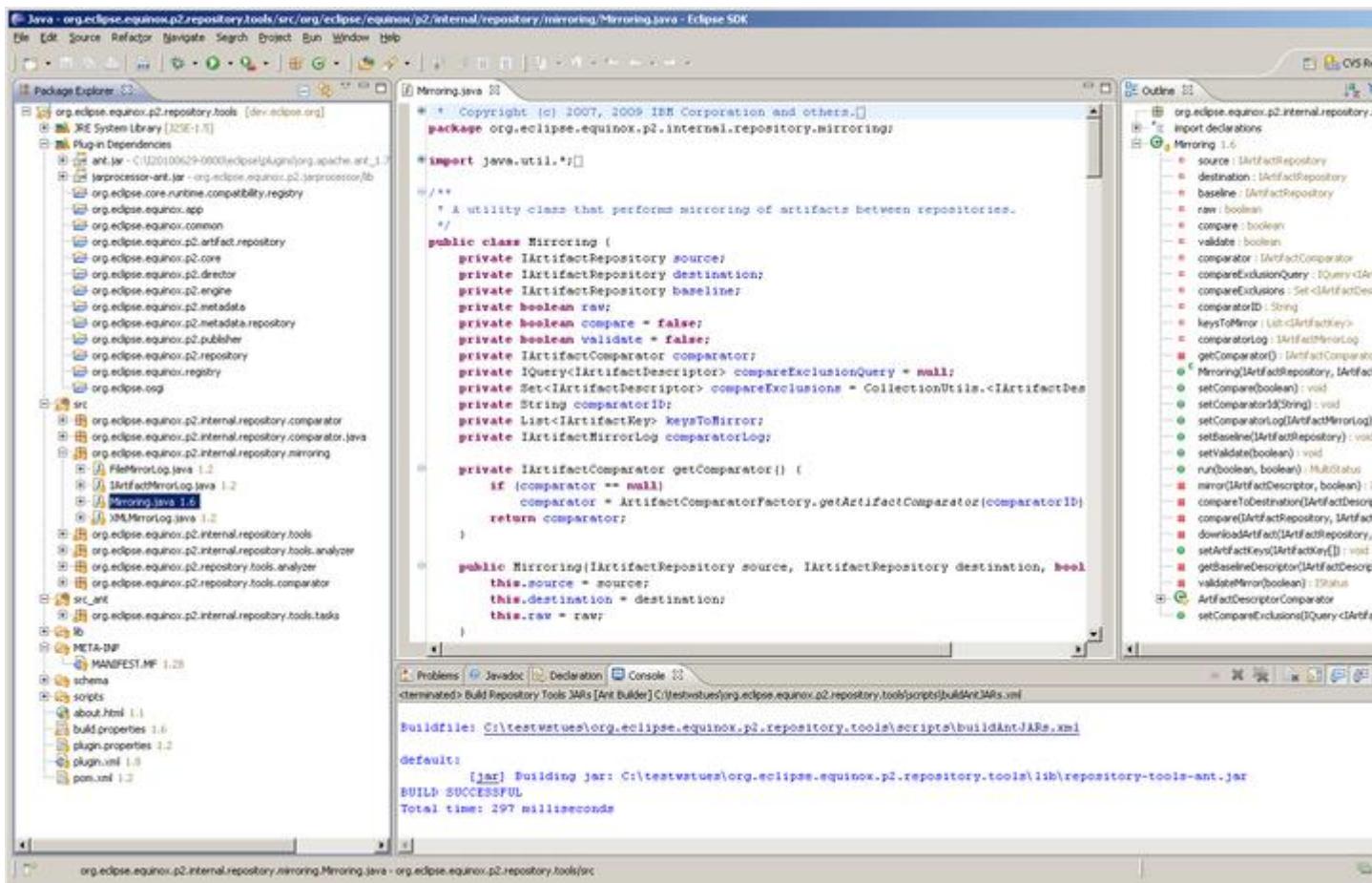


Рис.1: Набор панелей для работы с языком Java

В ранних вариантах архитектуры Eclipse SDK было три основных элемента, которые соответствовали трем основным подпроектам: платформе, инструментальным средствам JDT (Java Development Tools) и среде разработки плагинов PDE (Plug-In Development Environment).

6.1.1. Платформа

Платформа Eclipse написана с использованием Java и для того, чтобы ее запустить, требуется виртуальная машина Java VM. Платформа построена из небольших функциональных блоков, называемых плагинами. Плагины являются основой компонентной модели Eclipse. Плагин является, по существу, JAR-файлами с манифестом, в котором описывается плагин, его зависимости, и то, как его можно использовать или расширять. Эта информация манифеста изначально хранилась в файле `plug-in.xml`, который находился в корневой папке плагина. Инструментальные средства Java представляют собой плагины, предназначенные для ведения разработки на языке Java. Среда разработки плагинов (PDE) предоставляет собой набор инструментов для разработки плагинов расширений Eclipse. Плагины Eclipse написаны на языке Java, но также могут содержать другие добавления, например, файлы HTML в качестве онлайновой документации. Каждый плагин может иметь свой собственный загрузчик классов. Плагины могут зависеть от других плагинов, что указывается в инструкциях `requires` в файле `plugin.xml`. Если взгляните на файл `plugin.xml` плагина `org.eclipse.ui`, вы увидите его название и номер версии, а также зависимости, которые необходимо импортировать из других плагинов.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
    id="org.eclipse.ui"
    name="%Plugin.name"
    version="2.1.1"
    provider-name="%Plugin.providerName"
    class="org.eclipse.ui.internal.UIPlugin">
```

```

<runtime>
  <library name="ui.jar">
    <export name="*"/>
    <packages prefixes="org.eclipse.ui"/>
  </library>
</runtime>
<requires>
  <import plugin="org.apache.xerces"/>
  <import plugin="org.eclipse.core.resources"/>
  <import plugin="org.eclipse.update.core"/>
  :
  :
  <import plugin="org.eclipse.text" export="true"/>
  <import plugin="org.eclipse.ui.workbench.texteditor" export="true"/>
  <import plugin="org.eclipse.ui.editors" export="true"/>
</requires>
</plugin>

```

Чтобы поощрять создание проектов на платформе Eclipse, требуется механизм, который может расширять платформу, и нужно, чтобы платформа могла использовать такое расширение. Это достигается за счет использования расширений и точек расширений, еще одного элемента компонентной модели Eclipse. С помощью экспорта определяются интерфейсы, которыми, как вы ожидаете, воспользуются другие разработчики, когда будут писать собственные расширения; в результате классы, доступные за пределами вашего плагина, ограничиваются только теми, которые экспортятся. Также устанавливаются дополнительные ограничения на ресурсы, доступные за пределами плагина, что представляет собой отличие от того, когда доступными делаются все публичные методы или классы (т. е. типа public — прим.пер.). Экспортируемые плагины рассматриваются как публичный интерфейс API. Все остальное считается частными особенностями реализации (т. е. типа private — прим.пер.). Чтобы написать плагин, который добавит пункт меню на панели инструментов Eclipse, вы можете использовать точку расширения `actionSets` в плагине `org.eclipse.ui`.

```

<extension-point id="actionSets" name="%ExtPoint.actionSets"
                 schema="schema/actionSets.exsd"/>
<extension-point id="commands" name="%ExtPoint.commands"
                 schema="schema/commands.exsd"/>
<extension-point id="contexts" name="%ExtPoint.contexts"
                 schema="schema/contextes.exsd"/>
<extension-point id="decorators" name="%ExtPoint.decorators"
                 schema="schema/decorators.exsd"/>
<extension-point id="dropActions" name="%ExtPoint.dropActions"
                 schema="schema/dropActions.exsd"/>

```

Расширение вашего плагина, которое добавляет пункт меню к точке расширения `org.eclipse.ui.actionSet`, будет выглядеть следующим образом:

```

<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="com.example.helloworld"
  name="com.example.helloworld"
  version="1.0.0">
  <runtime>
    <library name="helloworld.jar"/>
  </runtime>
  <requires>
    <import plugin="org.eclipse.ui"/>
  </requires>
  <extension
    point="org.eclipse.ui.actionSets">
    <actionSet
      label="Example Action Set"
      visible="true"
      id="org.eclipse.helloworld.actionSet">

```

```

<menu
    label="Example &Menu"
    id="exampleMenu">
    <separator
        name="exampleGroup">
    </separator>
</menu>
<action
    label="&Example Action"
    icon="icons/example.gif"
    tooltip="Hello, Eclipse world"
    class="com.example.helloworld.actions.ExampleAction"
    menuBarPath="exampleMenu/exampleGroup"
    toolbarPath="exampleGroup"
    id="org.eclipse.helloworld.actions.ExampleAction">
</action>
</actionSet>
</extension>
</plugin>

```

Когда запускается Eclipse, среда выполнения платформы сканирует манифесты плагинов вашей инсталляции и строит реестр плагинов, хранящихся в памяти. Отображения между точками расширений и соответствующими расширениями осуществляются по именам. Получившийся в результате реестр плагинов можно использовать из интерфейса API, предоставляемого платформой Eclipse. Реестр кэшируется на диске с тем, чтобы эту информацию могла было загрузить снова в следующий раз, когда Eclipse будет перезапущен. Все плагины, которые будут обнаружены при запуске, будут занесены в реестр, но они не будут активированы (загруженные классы) до тех пор, пока код не будет в действительности использован. Этот подход называется ленивой или отложенной активацией. Влияние добавления дополнительных сборок на производительность вашей инсталляции уменьшается за счет того, что действительная загрузка классов, ассоциированных с плагинами, не будет происходить до тех пор, пока классы действительно не понадобятся. Например, плагин, который добавляется к точке расширения org.eclipse.ui.actionSet, не будет активироваться до тех пор, пока пользователь не выберет новый пункт меню на панели инструментов.



Рис.6.2: Пример меню

Код, с помощью которого создается этот пункт меню, выглядит следующим образом:

```

package com.example.helloworld.actions;

import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.eclipse.jface.dialogs.MessageDialog;

public class ExampleAction implements IWorkbenchWindowActionDelegate {
    private IWorkbenchWindow window;

    public ExampleAction() {
    }

    public void run(IAction action) {
        MessageDialog.openInformation(
            window.getShell(),
            "org.eclipse.helloworld",
            "Hello, Eclipse architecture world");
    }
}

```

```

}

public void selectionChanged(IAction action, ISelection selection) {

}

public void dispose() {

}

public void init(IWorkbenchWindow window) {
    this.window = window;
}
}

```

Как только пользователь выбирает новый элемент на панели инструментов, плагин, реализующий точку расширения, делает запрос в регистр расширений. Плагин, реализующий расширение, создает экземпляр расширения и плагин загружается. Как только будет активирован плагин, в нашем примере будет запущен конструктор `ExampleAction`, а затем будет инициализировано делегируемое действие `Workbench`. Поскольку выбор в рабочем пространстве изменился и был создан делегат, действие может поменяться. Откроется диалоговое окно с сообщением «Hello, Eclipse architecture world».

Такая расширяемая архитектура была одним из ключей к успешному росту экосистемы Eclipse. Компании или частные лица могли разрабатывать новые плагины и либо реализовывать их в виде открытого исходного кода или продавать их как коммерческие изделия.

Одна из наиболее важных концепций, касающаяся Eclipse, представляет собой утверждение, что *все является плагинами*. Независимо от того, входит ли плагин в платформу Eclipse, или вы написали его самостоятельно, все плагины являются главными компонентами собранного приложения. На рис.6.3 показаны кластеры функциональных возможностей, предоставляемых плагинами в ранних вариантах Eclipse.

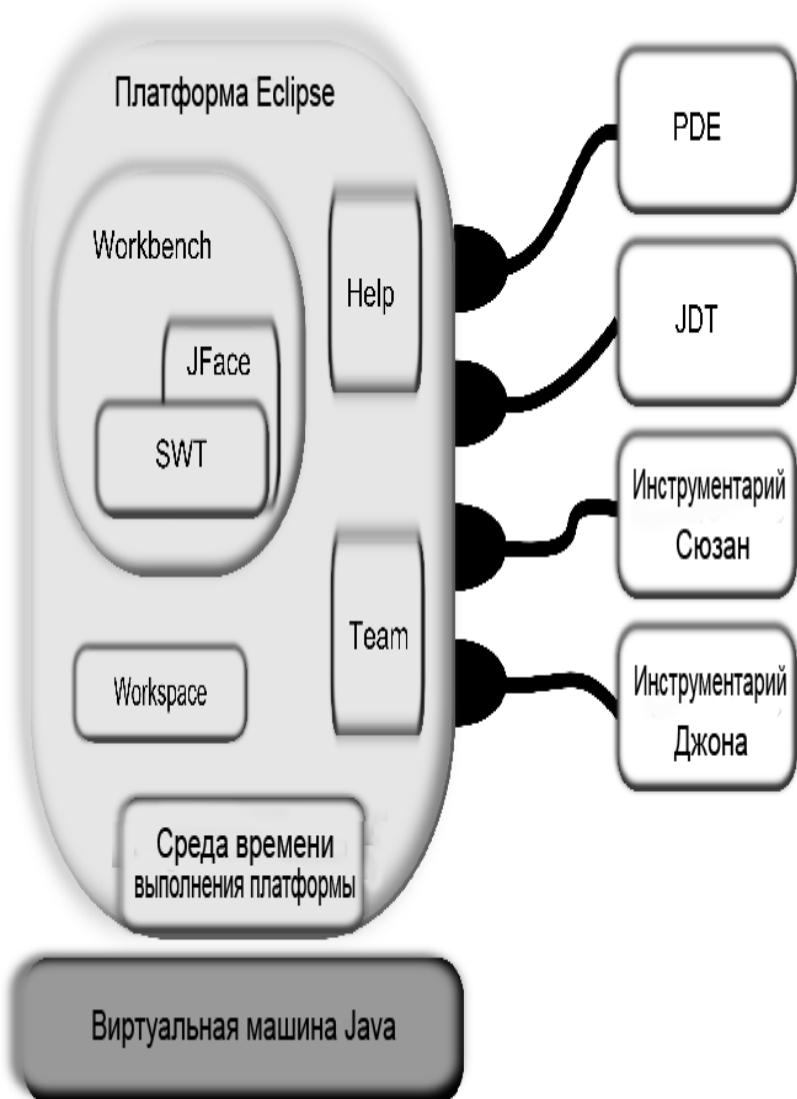


Рис.6.3: Архитектура ранних вариантов Eclipse

Рабочее пространство (workbench) является наиболее известным элементом интерфейса для пользователей платформы Eclipse, т. к. в нем находятся данные, определяющие, как Eclipse будет представлен пользователю на рабочем столе. Рабочее пространство состоит из наборов окон, отдельных окон и окон редакторов. Редакторы ассоциированы с типами файлов и, поэтому, при открытии файла запускается необходимый редактор. Примером отдельного окна является окно «problems» (Проблемы), в котором указываются ошибки или предупреждения, касающиеся вашего кода на Java. Окна редакторов и отдельные окна совместно образуют набор окон или перспективу (perspective), в котором пользователям предоставляются инструментальные средства, организованные определенным образом.

Рабочее пространство Eclipse строится на базе инструментального набора виджетов Standard Widget Toolkit (SWT) и Jface, причем SWT заслуживает немного более детального рассмотрения. Инструментальные наборы виджетов могут быть как нативными, так и эмулирующими. Нативный инструментальный набор виджетов использует для создания компонентов пользовательского интерфейса, например, списков и кнопок, обращения к операционной системе. Взаимодействие с компонентами обрабатывается операционной системой. Эмулирующий инструментальный набор виджетов реализует компонент без обращения к операционной системе и самостоятельно обрабатывает нажатия мыши и клавиатуры, рисует, управляет фокусом и другими функциями и не перекладывает все это на операционную систему. Обе конструкции имеют свои сильные и слабые стороны.

Нативные инструментальные наборы виджетов являются «самим совершенством». Их виджеты выглядят на рабочем столе также, как и их аналоги из других приложений. Поставщики операционных систем постоянно изменяют внешний вид своих виджетов и добавляют новые возможности. Нативные инструментальные наборы виджетов получают эти обновления без всяких дополнительных затрат. К сожалению, нативные инструментальные наборы трудно реализовывать, поскольку виджеты операционной системы, на базе которых они создаются, значительно различаются и приводят к противоречиям и программы теряют свойство переносимости.

Эмулирующий инструментальный набор виджетов либо имеет свой собственный внешний вид, либо пытается выполнять отрисовку и вести себя точно так, как это происходит в операционной системе. Их главное преимущество над нативными наборами состоит в их гибкости (хотя современные нативные инструментальные наборы виджетов, например, Windows Presentation Framework (WPF), столь же гибки). Поскольку код, реализующий виджет, является частью инструментария, а не встроен в операционную систему, виджет может выполнять отрисовку и вести себя любым образом. Программы, в которых используются эмулирующие инструментальные наборы виджетов, хорошо переносятся одной системы на другие. Ранние варианты эмулирующих инструментальных наборов виджетов имели плохую репутацию. Часто они были медлительными и плохо эмулировали работу операционной системы и из-за этого они выглядели неуместно на рабочем столе. В частности, в то время можно было легко отличить программы на языке Smalltalk-80 из-за того, что в них использовались эмулирующие виджеты. Пользователи знали, что они управляли «программой Smalltalk» и это плохо влияло на принятие приложений, написанных на Smalltalk.

В отличие от других языков программирования, например, С и С++, первые версии языка Java были доступны с нативным инструментальным набором виджетов, который имел название Abstract Window Toolkit (AWT). Считается, что AWT достаточно ограниченный, имеет ошибки, противоречивый и все его ругают. Фирма Sun и многие другие, отчасти из-за того, что имели опыт работы с AWT, считали нативный инструментальный набор виджетов, который был переносимым и обладал хорошей производительностью, непригодным для работы. Решением был Swing, полнофункциональный эмулирующий инструментальный набор виджетов.

Приблизительно в 1999 году подразделение OTI использовало язык Java для реализации продукта под названием VisualAge Micro Edition. В первой версии VisualAge Micro Edition использовался Swing, причем опыт OTI по использованию Swing не был положительным. Ранние версии Swing имели ошибки, работали долго и тратили много памяти, а аппаратура того времени была недостаточно мощной с тем, чтобы обеспечить приемлемую производительность. Подразделение OTI успешно создало нативный инструментальный набор виджетов для Smalltalk-80 и для других реализаций Smalltalk, в результате чего язык Smalltalk был встречен с одобрением. Этот опыт был использован для создания первой версии пакета SWT. VisualAge Micro Edition и SWT имели успех и когда началась работа над Eclipse, естественным выбором стал SWT. Использование в Eclipse пакета SWT поверх Swing раскололо сообщество Java. Некоторые видели заговор, но Eclipse добился успеха и использование SWT отличало его от других программ на языке Java. Eclipse обладал достаточной производительностью, имел хороший внешний вид и общее настроение было следующим: «Я не могу поверить, что это программа на языке Java».

Ранние варианты Eclipse SDK работали на Linux и Windows. В 2010 году появилась поддержка более десятка платформ. Разработчик может написать приложение для одной платформы, и развернуть его на нескольких платформах. В то время в рамках сообщества Java разработка нового инструментального набора виджетов для Java считалась спорным вопросом, но те, кто работал с Eclipse, чувствовали, что нужно потратить усилия и получить лучшее нативное решение, используемое на рабочем столе. Это справедливо и сегодня, и есть миллионы строк кода, зависящего от SWT.

JFace представляет собой слой поверх SWT, предоставляющий инструментальные средства для обычных задач программирования пользовательского интерфейса, например, фреймворки для ра-

боты с настройками и визардами. Точно также, как и SWT, он разрабатывался так, чтобы его можно было использовать со многими оконными системами. Тем не менее, это чистый код на языке Java, в котором нет нативного кода платформ.

В рамках платформы также предоставляется интегрированная система подсказок, базирующаяся на небольших блоках информации, которые называются темами. Тема состоит из метки и ссылки на место, где находится сама тема. Ссылка может указывать на HTML-файл с документацией или на XML-документ, описывающий дополнительные ссылки. Темы группируются вместе в виде оглавлений (в TOC). Темы можно рассматривать как листья, а TOC - как три подразделения организации. Для того, чтобы добавить содержимое подсказки к вашему приложению, вы можете к точке расширения `org.eclipse.help.toc` добавить `org.eclipse.platform.doc.isv plugin.xml` так, как это сделано ниже.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>

<!-- ===== -->
<!-- Определяем первичное содержимое TOC -->
<!-- ===== -->
&l;textension
    point="org.eclipse.help.toc">
    <toc
        file="toc.xml"
        primary="true">
    </toc>
    <index path="index"/>
</extension>
<!-- ===== -->
<!-- Определяем содержимое TOC для подтем -->
<!-- ===== -->
<extension
    point="org.eclipse.help.toc">
    <;toc
        file="topics_Guide.xml">
    </toc>
    &l;ttoct
        file="topics_Reference.xml">
    </toc>
    <toc
        file="topics_Porting.xml">
    </toc>
    <toc
        file="topics_Questions.xml">
    &l;t/toc>
    <toc
        file="topics_Samples.xml">
    </toc>
</extension>
```

Для индексирования и онлайнового поиска содержимого подсказок используется пакет Apache Lucene. В ранних вариантах Eclipse онлайновая подсказка представляла собой веб-приложение, работающая на сервере Tomcat. Кроме того, за счет предоставления подсказки в самом Eclipse, у вас также могли использовать некоторое подмножество плагинов, предназначенные для работы с подсказками, для того, чтобы создавать автономно работающие серверы подсказок [3].

Eclipse также предоставлял поддержку среди командной работы с репозитарием исходного кода, создания патчей и других обычно выполняемых задач. В рабочем пространстве предоставлялся набор файлов и метаданных, с помощью которых ваша работа будет сохранена в файловой системе. Имелся также отладчик, позволяющий отслеживать проблемы в коде Java, а также фреймворк для построения отладчиков для конкретных языков.

Одна из целей проекта Eclipse состояла в поддержке разработчиков программ с открытым исходным кодом и разработчиков коммерческих программ в использовании данной технологии для расширения платформы в соответствие с их собственными потребностями, причем один из способов поощрения применения данной технологии заключался в предоставлении стабильного интерфейса API. API можно рассматривать как технический контракт, определяющий поведение вашего приложения. Его также можно рассматривать как социальный контракт. В проекте Eclipse есть манtra - «API - навсегда». Так что следует быть исключительно аккуратным когда пишется интерфейс API и учитывать, что он предназначен для использования на неопределенно долгий срок. Стабильное API представляет собой контракт между клиентом или потребителем и поставщиком API. Этот контракт гарантирует, что клиент может положиться на использование платформы Eclipse, API в которой предоставляет на длительный срок, и не надо будет беспокоиться о том, что потребуется рефакторинг части клиентского приложения. Хороший интерфейс API также достаточно гибок с тем, чтобы можно было развивать его реализации.

6.1.2. Инструментарий Java Development Tools (JDT)

В JDT предоставляются редакторы языка Java, визарды, средства поддержки рефакторинга, отладчик, компилятор и инкрементный сборщик. Компилятор также помогает в наборе кода, навигации по коду и имеет другие функции редактирования. Java SDK не поставляется вместе с Eclipse, так что от пользователя зависит, какой SDK он установит на своем компьютере. Почему команда, разрабатывающая JDT, написала отдельный компилятор для компиляции Java-кода в Eclipse? Первоначально был взят код компилятора из проекта VisualAge Micro Edition. Планировалось поверх компилятора создать инструментальные средства, так что написание самого компилятора было логичным решением. Этот подход также позволил разработчикам JDT создавать точки расширения с тем, чтобы можно было расширять компилятор. Это было бы трудно делать, если бы компилятор был работающим из командной строки приложением, предоставляемым третьей стороной.

Написание своего собственного компилятора позволяет иметь средство, которое обеспечивает поддержку инкрементного сборщика, имеющегося в IDE. Инкрементный сборщик позволяет улучшить производительность, т. к. он заново компилирует только те файлы или их зависимости, которые были изменены. Как работает инкрементный сборщик? Когда вы в Eclipse создаете проект на языке Java, вы создаете ресурсы в рабочем пространстве, в котором хранятся ваши файлы. Сборщик в Eclipse берет входные файлы из вашего рабочего пространства (файлы .java) и создает выходные файлы (файлы .class). По состоянию процесса сборки сборщик знает о типах (классах или интерфейсах), имеющихся в рабочем пространстве, и то, как они соотносятся друг с другом. Состояние процесса сборки сообщается сборщику компилятором каждый раз, когда компилируется некоторый исходный файл. Когда выполняется инкрементная сборка, сборщику сообщается только об изменениях в ресурсах, т. е. сообщается о всех новых, измененных или удаленных файлах. Если файлы удаляются, то удаляются соответствующие файлы классов. Новые или измененные типы добавляются в очередь. Файлы в очереди компилируются один за другим и сравниваются со старым файлом класса для того, чтобы определить, были ли структурные изменения. Структурные изменения представляют собой модификации класса, которые могут влиять на другой тип, ссылающийся на него, например, изменение сигнатуры метода, добавление или удаление метода. Если происходят структурные изменения, то все типы, которые ссылаются на него, также добавляются в очередь. Если тип вообще был изменен, то новый файл класса записывается в каталог выходных файлов сборщика. Состояние процесса сборки обновляется в соответствие с информацией о скомпилированном типе. Этот процесс повторяется для всех типов в очереди до тех пор, пока очередь не будет пуста. Если есть ошибки компиляции, редактор языка Java создаст маркеры, указывающие на проблему. На протяжении многих лет инструментальные средства, предоставляемые в составе JDT, постоянно расширялись согласованно с появлением новых версий самой среды выполнения Java.

6.1.3. Среда разработки плагинов PDE

В среде разработки плагинов Plug-in Development Environment (PDE) предоставляются инструментальные средства для разработки, сборки, установки и тестирования плагинов и других артефактов, которые используются для расширения функциональных возможностей платформы Eclipse. Поскольку плагины Eclipse были новым типом артефактов в мире Java, не было системы сборки, которая могла бы преобразовывать исходный код в плагины. Поэтому команда разработчиков PDE написала компонент, называемый Сборщик PDE (PDE Build), который проверял зависимости плагинов и создавал скрипты Ant для сборки этих артефактов.

6.2. Eclipse 3.0: Среда времени выполнения, RCP и роботы

6.2.1. Среда времени выполнения

Eclipse 3.0 был, вероятно, одним из самых важных релизов проекта Eclipse благодаря ряду существенных изменений, которые произошли в течение этого цикла выпуска. В архитектуре Eclipse, которая предваряла версию 3.0, компонентная модель Eclipse состояла из плагинов, которые могли взаимодействовать друг с другом двумя способами. Во-первых, зависимости между ними могли быть выражены при помощи инструкции `requires` в файлах `plugin.xml`. Если плагину A требуется плагин B, то плагин A может видеть все Java классы и ресурсы из B, учитывая, конечно, соглашения о видимости классов языка Java. Каждый плагин имел версию, так что также можно было указывать версии зависимостей. Во-вторых, компонентная модель предлагала использовать *расширения и точки расширения*. Исторически сложилось, что разработчики, использующие Eclipse, написали свою собственную среду выполнения для Eclipse SDK, которая управляла загрузкой классов, зависимостями плагинов и расширениями и точками расширений.

Проект Equinox был создан как новый инкубационный проект в рамках Eclipse. Целью проекта Equinox была замена компонентной модели Eclipse тем, что уже существовало, а также возможность обеспечить поддержку динамических плагинов. Среди рассматриваемых решений были JMX, Jakarta Avalon и OSGi. JMX не была полностью разработанной компонентной моделью и поэтому было решено, что ее использовать нецелесообразно. Jakarta Avalon не был выбран потому, что, как оказалось, он уже перестал существовать как проект. В дополнение к техническим требованиям, также было важно учитывать сообщество, которое поддерживает конкретную технологию. Будет ли сообщество готовы принять изменения, связанные с Eclipse? Готовы ли оно активно развиваться и расширяться в новых условиях? Команда разработчиков Equinox понимала, что сообщество, сгруппированное вокруг технологии, которую они в конце концов выберут, столь же важно, как и технические соображения.

После исследования и оценки имеющихся альтернатив, разработчики выбрали проект OSGi. Почему проект OSGi? В нем была семантическая схема именования версий, используемая для управления зависимостями. Это позволяло воспользоваться модульным фреймворком, которого не хватало в самом JDK. Пакеты, которые предоставлялись для других сборок, должны были явно экспортirоваться, а все остальные пакеты были скрыты. В OSGi был свой собственный загрузчик классов, поэтому команде разработчиков Equinox не требовалось продолжать поддерживать свои собственные загрузчики. Благодаря тому, что была стандартизирована компонентная модель, распространение которой не ограничивалось только экосистемой Eclipse, было понятно, что можно было обращаться к более широкой аудитории, и шире распространять проект Eclipse.

Команда разработчиков Equinox чувствовала себя уверенно, т. к. у проекта OSGi уже было энергичное сообщество; и они могли работать с этим сообществом, что помогло добавить функциональные возможности компонентной модели, необходимые проекту Eclipse. Например, на тот момент в OSGi поддерживались требования делать перечисления свойств на уровне пакетов, а не на уровне плагинов, как это требовалось в Eclipse. Кроме того, в OSGi на тот момент еще не использовалась концепция фрагментов, которая в Eclipse являлась предпочтительным механизмом добавления кода, специфического для конкретной платформы или среды окружения, в существующий плагин. Например, фрагменты кода, предоставляемые для работы в файловых системах Linux

и Windows, а также фрагменты, предназначенные для трансляции на другие языки. Как только было принято решение в качестве новой среды выполнения начать использовать OSGi, разработчикам потребовалась реализация фреймворка с открытым исходным кодом. Был проанализирован проект Oscar, который был предшественником проекта Apache Felix, а также фреймворк управления сервисами Service Management Framework (SMF), разработанный фирмой IBM. На тот момент Оскар был исследовательским проектом с ограниченными возможностями внедрения. В конечном итоге был выбран проект SMF, т.к. он уже использовался в поставляемых изделиях и, таким образом, был признан в качестве используемого в среде уровня предприятиями. За эталонную реализацию спецификации OSGi была взята реализация Equinox.

Был также реализован слой совместимости, так что в версии 3.0 по-прежнему должны работать уже существующие плагины. Просить разработчиков переписать их плагины согласно изменениями в базовой инфраструктуре Eclipse 3.0 было бы для проекта Eclipse, как инструментальной платформы, импульсом в направлении тупика. По ожиданиям разработчиков, использующих Eclipse, платформа должна просто продолжать работать.

С переходом на OSGi, плагины Eclipse стали еще называться сборками. Плагин и сборка являются одним и тем же: они оба обеспечивают модульное подмножество функциональных возможностей, которое описывается с помощью метаданных в манифесте. Раньше зависимости, экспортруемые пакеты и расширения, а также точки расширения описывались в файле plugin.xml. С переходом на сборки OSGi, описания расширений и точек расширений продолжали указываться в файле plugin.xml, поскольку они относятся к понятиям проекта Eclipse. Остальная информация описывалась в файле META-INF/MANIFEST.MF, т. е. в манифесте сборки версии OSGi. Чтобы поддержать такое изменение, в PDE был предложен новый редактор манифестов для использования внутри Eclipse. Каждая сборка имеет имя и версию. Манифест для сборки org.eclipse.ui выглядит следующим образом:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: %Plugin.name
Bundle-SymbolicName: org.eclipse.ui; singleton:=true
Bundle-Version: 3.3.0.qualifier
Bundle-ClassPath: .
Bundle-Activator: org.eclipse.ui.internal.UIPlugin
Bundle-Vendor: %Plugin.providerName
Bundle-Localization: plugin
Export-Package: org.eclipse.ui.internal;x-internal:=true
Require-Bundle: org.eclipse.core.runtime;bundle-version="[3.2.0,4.0.0)",
  org.eclipse.swt;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
  org.eclipse.jface;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
  org.eclipse.ui.workbench;bundle-version="[3.3.0,4.0.0)";visibility:=reexport,
  org.eclipse.core.expressions;bundle-version="[3.3.0,4.0.0)"
Eclipse-LazyStart: true
Bundle-RequiredExecutionEnvironment: CDC-1.0/Foundation-1.0, J2SE-1.3
```

Начиная с Eclipse 3.1, в манифесте также можно определять среду выполнения, необходимую для работы сборки (bundle required execution environment - BREE). Среда выполнения указывалась как минимальная среда Java, необходимая для работы сборки. Компилятор Java не умеет разбираться в сборках и манифестах OSGi. В PDE есть инструментальные средства для разработки сборок OSGi. Поэтому PDE выполняет анализ манифеста сборки и создает для этой сборки путь classpath. Если вы в своем манифесте указали в качестве среды исполнения среду J2SE-1.4, а затем написали код, в котором есть обобщенные типы generic, то вам будет предложено исправить ошибки кода. Тем самым гарантируется, что ваш код соответствует контракту, указанному в манифесте.

В OSGi предоставляется модульный фреймворк для языка Java. Фреймворк OSGi осуществляет управление коллекциями самодокументированных сборок и загрузкой их классов. В каждом пакете есть свой собственный загрузчик классов. Путь classpath, используемый в сборке, строится путем проверки зависимостей, указанных в манифесте, и собственно генерации пути classpath. При-

ложения OSGi являются коллекциями сборок. Для того, чтобы в полной мере воспользоваться модульностью, вы должны уметь выразить зависимости вашего приложения в виде, понятном для потребителей. Поэтому в манифесте описываются экспортируемые пакеты, доступные для клиентов этой сборки, которые представляют собой общедоступный интерфейс API, предоставляемый для использования. В сборке, в которой используется этот интерфейс API, должен быть соответствующий импорт используемых пакетов. Манифест также позволяет вам указать диапазоны версий ваших зависимостей. Взгляните на инструкцию `Require-Bundle`, которая имеется в приведенном выше манифесте, и обратите внимание на то, что сборка `org.eclipse.core.runtime`, от которой зависит `org.eclipse.ui`, должна иметь версию не меньше, чем 3.2.0 и не больше, чем 4.0.0.

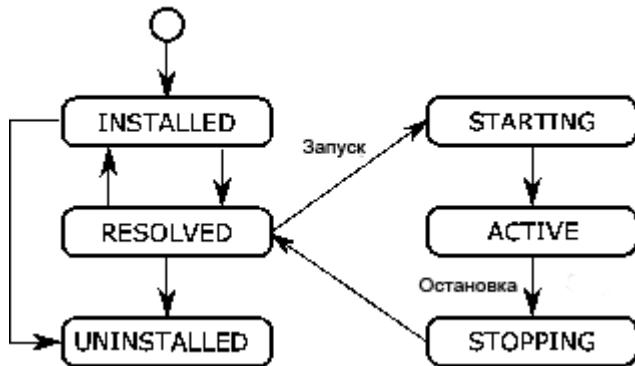


Рис.6.4: Жизненный цикл сборки OSGi

OSGi является динамическим фреймворком, который поддерживает установку, запуск, остановку и удаление сборок. Как упоминалось ранее, одним из основных преимуществ Eclipse является ленивая или отложенная активация, когда классы плагинов не загружаются до тех, пока они не становятся необходимыми. Жизненный цикл сборок OSGi также позволяет использовать этот подход. Когда вы запускаете приложение OSGi, сборки находятся в состоянии `installed` или «установлено». Если зависимости сборки разрешены, то сборка переходит в состояние `resolved` или «зависимости разрешены». Как только зависимости будут разрешены, классы, находящиеся внутри этой сборки, могут загружаться и выполняться. Состояние `starting` или «запуск» означает, что сборка активизируется в соответствии с ее политикой активации. После того, как сборка активирована (т. е. находится в состоянии `active` или «активном» состоянии) и может запрашивать необходимые ресурсы и взаимодействовать с другими сборками. Сборка находится в состоянии `stopping` или «остановлено», когда выполняется метод `stop` ее активатора с тем, чтобы освободить все ресурсы, которые были открыты, когда сборка была активной. Наконец, сборка может быть «удалена» (т. е. находится в состоянии `uninstalled`), а это значит, что она будет недоступна для использования.

Т.к. интерфейс API развивается, то нужен способ сообщать потребителям вашего API об изменениях. Один подходов состоит в использовании семантической схемы именования версий ваших сборок и указания в манифестах диапазонов версий для вашей зависимости. В OSGi используется схема именования версий, имеющая четыре части и показанная на рис.6.5.

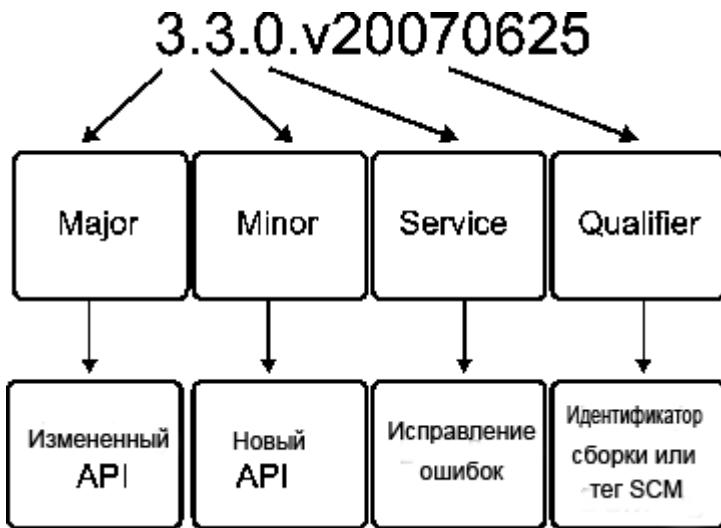


Рис.6.5: Схема именования версий

В схеме нумерации версий OSGi, каждая сборка имеет уникальный идентификатор, состоящий из имени и четырех частей номера версии. Идентификатор и версия являются вместе для потребителя уникальным набором байтов. Согласно правилам Eclipse, если вы делаете изменения в сборке, то каждая часть номера версии будет указывать потребителям вид сделанных изменений. Таким образом, если вы хотите указать, что вы намереваетесь изменить API, вы на единицу увеличиваете первую (major) часть номера версии. Если вы просто расширили API, вы увеличиваете на единицу вторую часть (minor) номера версии. Если вы исправили небольшую ошибку, которая не влияет на API, то на единицу увеличивается третья часть (service) номера версии. Наконец, когда на единицу увеличивается четвертая часть номера (qualifier), то он указывает на новый идентификатор собранного варианта или тег репозитория с исходным кодом.

Кроме возможности определять фиксированные зависимости между сборками, в OSGi также есть механизм, называемый сервисами, который позволяет реализовывать дополнительную связь между сборками. Сервисами являются объекты с набором свойств, которые регистрируются в реестре сервисов OSGi. В отличие от расширений, которые регистрируются в реестре расширений в тот момент, когда Eclipse сканирует сборки при запуске, сервисы регистрируются динамически. В сборку, в которой используется сервис, нужно импортировать пакет, определяющий сервис-контракт, а фреймворк определяет реализацию сервиса, взятого из реестра сервисов.

Есть определенное приложение, предназначенное для запуска Eclipse, которое похоже на метод main в файле классов Java. Приложения Eclipse определяются с помощью расширений. Например, приложением для запуска самого Eclipse IDE является приложение org.eclipse.ui.ide.workbench, которое определено в сборке org.eclipse.ui.ide.application.

```

<plugin>
    <extension
        id="org.eclipse.ui.ide.workbench"
        point="org.eclipse.core.runtime.applications">
        <application>
            <run
                class="org.eclipse.ui.internal.ide.application.IDEApplication">
            </run>
        </application>
    </extension>
</plugin>
  
```

Есть много приложений, предоставляемых в Eclipse, например, приложения для автономного запуска серверов подсказки, выполнения задач Ant и тестов JUnit.

6.2.2. Платформа Rich Client Platform (RCP)

Одна из самых интересных особенностей, касающаяся работы в сообществе, использующим открытый исходный код, состоит в том, что программы могут применяться совершенно неожиданным образом. Первоначальным назначением проекта Eclipse было предоставление платформы и инструментальных средств, позволяющих создавать и расширять различные IDE. Однако к тому времени, когда был выпущен релиз 3.0, по отчетам об ошибках стало ясно, что сообщество взяло некоторое количество сборок, предназначенных для разработки платформы, и использовало их для сборки многофункциональных приложений Rich Client Platform (RCP), которые многие рассматривают как приложения на языке Java. Т.к. первоначально Eclipse был ориентирован на создание IDE, то для того, чтобы новый вариант применения мог быть проще воспринят сообществом, потребовался определенный рефакторинг проекта. В приложениях RCP не требуются все функциональные возможности, которые нужны в IDE, так что некоторые сборки были разделены на более мелкие, которые могут использоваться сообществом при построении приложений RCP.

Примерами приложений RCP, используемых в реальных условиях, является применение платформы RCP для мониторинга роботов-марсоходов, разработанных НАСА в лаборатории Jet Propulsion Laboratory, использование проекта Bioclipse для визуализации данных биоинформатики и использование проекта Dutch Railway для мониторинга загруженности поездов. Общая мысль, проходящая через многие из этих приложений, заключается в том, что эти команды разработчиков решили, что могут взять утилиты, предлагаемые платформой RCP, и сконцентрироваться на создании своих специальных инструментальных средств, работающих поверх этих утилит. Они могут сэкономить время разработки и деньги благодаря тому, что могут сосредоточиться на создании своих инструментальных средств на базе платформы со стабильным интерфейсом API, что гарантирует, что технологии, выбранные ими, будут поддерживаться достаточно долго.

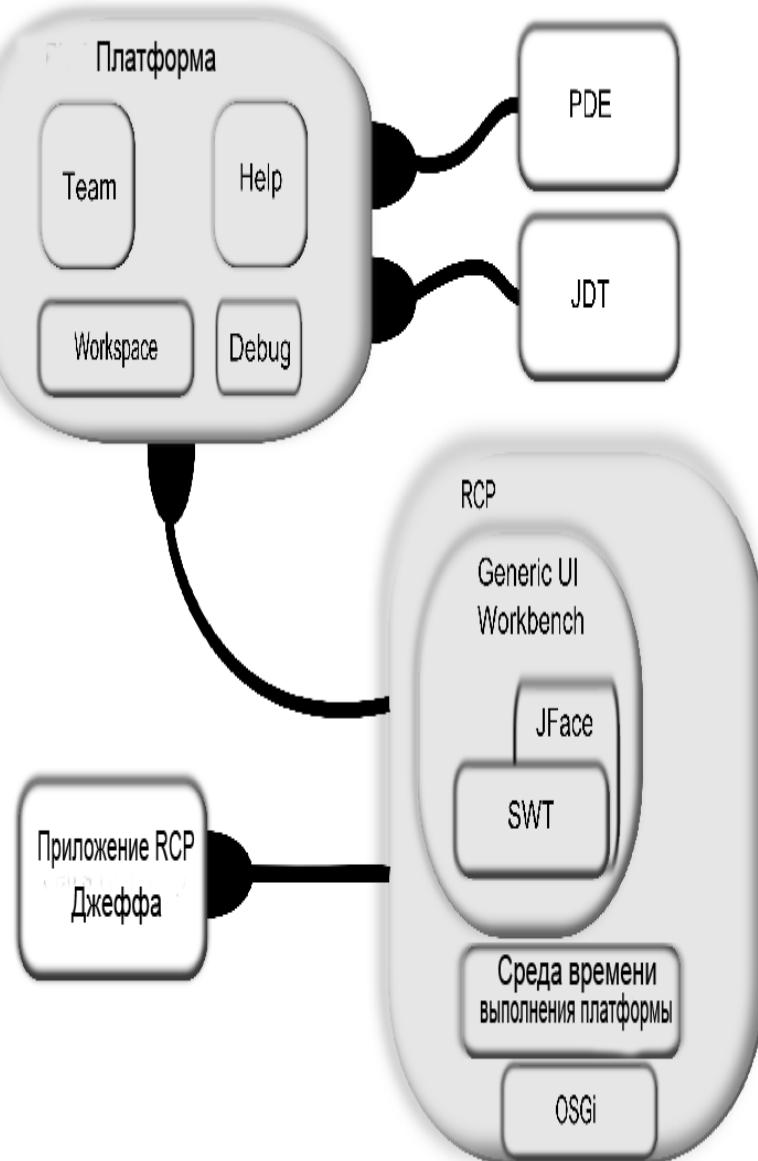


Рис.6.6: Архитектура Eclipse 3.0

Если вы посмотрите на архитектуру 3.0, приведенную на рис.6.6, то увидите, вы заметите, что для поддержки модели приложений и реестра расширений все еще присутствует среда времени выполнения Eclipse Runtime. Управление зависимостями между компонентами, т. е. Управление моделью плагинов, в настоящее время происходит при помощи OSGi. Кроме того, что пользователи могут продолжать расширять Eclipse с целью создания своих собственных сред разработки, они также могут на базе фреймворка приложений RCP собираять приложения более общего назначения.

6.3. Eclipse 3.4

Считается само собой разумеющейся возможность легкого обновления приложения до новой версии и добавление нового контента. В Firefox это происходит без проблем. Для Eclipse, это делать было не так просто. Первоначальным механизмом был менеджер обновлений (Update Manager), который использовался для добавления нового контента в инсталляцию Eclipse или обновление ее до новой версии.

Чтобы понять, что изменяется во время обновления или инсталляции, необходимо понять, что в Eclipse имеется в виду под понятием «возможности». Возможность является артефактом PDE, в котором определен набор пакетов, упаковываемых вместе в таком формате, который может быть

собран или установлен в виде одного релиза. Возможность может также включать в себя другие возможности. Смотрите рис.6.7.

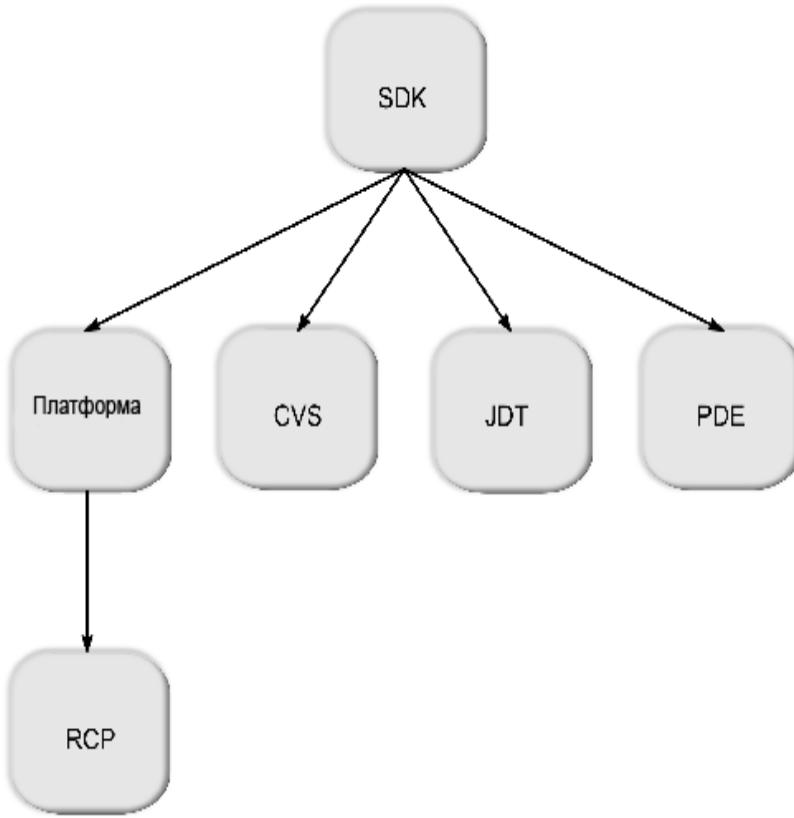


Рис.6.7: Иерархия возможностей Eclipse 3.3 SDK

Если вы хотели обновить свою инсталляцию Eclipse до нового релиза, в котором добавлен только один новый пакет, то из-за того что менеджер обновлений является достаточно грубым механизмом, потребуется обновить всю возможность целиком. Обновление возможности, в которой исправлен один пакет, является неэффективным.

Есть визарды PDE для создания возможностей и полученные с их помощью релизы помещаются в ваше рабочее пространство. В файле feature.xml указываются сборки, которые входят в состав конкретной возможности, а также указываются некоторые простые свойства сборок. Возможность, например, пакет, имеет имя и версию. В состав возможностей могут входить другие возможности., а также могут указываться диапазоны версий входящих возможностей. Пакеты, входящие в состав возможности, перечисляются с указанием конкретных свойств. Например, можно увидеть, что во фрагменте org.eclipse.launcher.gtk.linux.x86_64 указывается операционная система (os), система окон (ws) и архитектура (arch) той среды, где эту возможность можно использовать. Поэтому при обновлении до новой версии этот фрагмент будет устанавливаться только на указанной платформе. Такие фильтры платформ имеются в манифесте OSGi пакета.

```

<?xml version="1.0" encoding="UTF-8"?>
<feature
    id="org.eclipse.rcp"
    label="%featureName"
    version="3.7.0.qualifier"
    provider-name="%providerName"
    plugin="org.eclipse.rcp"
    image="eclipse_update_120.jpg">

    <description>
        %description
    </description>
  
```

```

<copyright>
    %copyright
</copyright>

<license url="%licenseURL">
    %license
</license>

<plugin
    id="org.eclipse.equinox.launcher"
    download-size="0"
    install-size="0"
    version="0.0.0"
    unpack="false"/>

<plugin
    id="org.eclipse.equinox.launcher.gtk.linux.x86_64"
    os="linux"
    ws="gtk"
    arch="x86_64"
    download-size="0"
    install-size="0"
    version="0.0.0"
    fragment="true"/>

```

Приложение Eclipse состоит не только из возможностей и сборок. Из следующего списка файлов, входящих в состав приложения Eclipse, видно, что есть платформозависимые исполняемые файлы, предназначенные для запуска самого Eclipse, файлы лицензий и платформозависимые библиотеки.

```

com.ibm.icu
org.eclipse.core.commands
org.eclipse.core.contenttype
org.eclipse.core.databinding
org.eclipse.core.databinding.beans
org.eclipse.core.expressions
org.eclipse.core.jobs
org.eclipse.core.runtime
org.eclipse.core.runtime.compatibility.auth
org.eclipse.equinox.common
org.eclipse.equinox.launcher
org.eclipse.equinox.launcher.carbon.macosx
org.eclipse.equinox.launcher.gtk.linux.ppc
org.eclipse.equinox.launcher.gtk.linux.s390
org.eclipse.equinox.launcher.gtk.linux.s390x
org.eclipse.equinox.launcher.gtk.linux.x86
org.eclipse.equinox.launcher.gtk.linux.x86_64

```

Эти файлы невозможно обновить с помощью менеджера обновлений, поскольку опять же, мы имеем дело только с возможностями. Поскольку многие из этих файлов обновлялись в каждом крупном релизе, это означало, что пользователи каждый раз, когда был новый релиз, должны были загружать новый файл zip, а не обновлять существующую инсталляцию. Это не подходило сообществу Eclipse. В PDE предоставлялась файловая поддержка, которая позволяла указывать все файлы, необходимые для создания приложений Eclipse RCP. Однако в менеджере обновлений не было механизма переноса этих файлов в вашу инсталляцию, что в равной мере разочаровывало как пользователей, так и разработчиков. В марте 2008 года в SDK появился компонент p2, представляющий собой новое решение, предназначенное для работы с ресурсами. Что обеспечить обратную совместимости, по-прежнему можно использовать менеджер обновлений, но по умолчанию применяется p2.

6.3.1. Концепции p2

В Equinox p2 все связано с инсталляционными единицами (installation units - IU). IU представляет собой определение имени и идентификатора артефакта, который вы устанавливаете. В этих метаданных также описываются возможности артефакта (что предлагается) и его требования (его зависимости). В метаданных также указывается фильтры применимости (область применения) в случае, если артефакт можно применять только в определенной среде. Например, фрагмент org.eclipse.swt.gtk.linux.x86 можно применять только в случае, если вы устанавливаете его на машине x86 с системой Linux и GTK. По сути, метаданные является формой представления информации, которая есть в манифесте сборки. Артефакты являются просто двоичными модулями, которые устанавливаются. Такое разделение достигается путем отделения метаданных от описываемых с их помощью артефактов. Репозиторий p2 состоит из репозитория метаданных и репозитория артефактов.

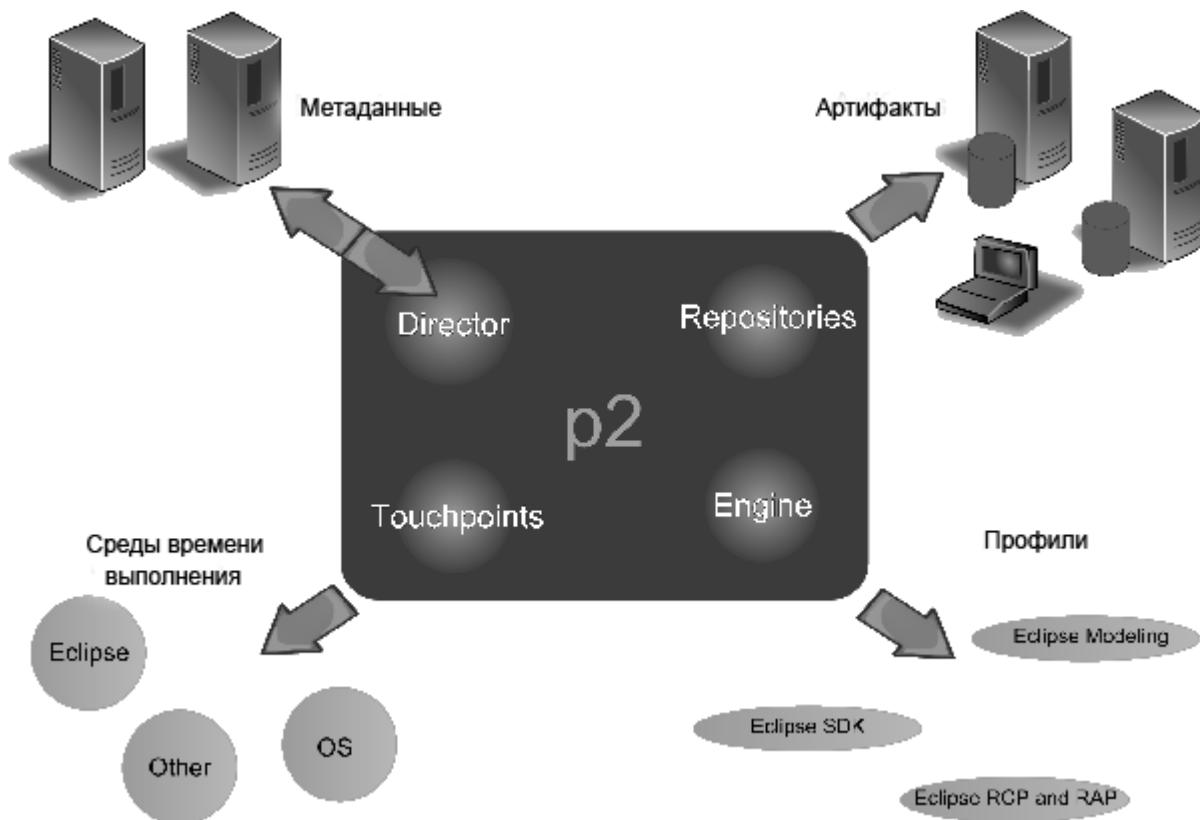


Рис.6.8: Концепции системы p2

Профиль представляет собой список фрагментов IU в вашем варианте инсталляции. Например, ваш Eclipse SDK имеет профиль, который описывает вашу текущую инсталляцию. Внутри Eclipse вы можете запросить обновление до более новой версии сборки, для которой будет создан новый профиль с другим набором фрагментов IU. В профиле также указывается список свойств, связанных с инсталляцией, например, операционная система, система управления окнами и параметры архитектуры. В профилиях также запоминается каталог инсталляции и место, где он находится. Профили находятся в реестре профилей, в котором могут храниться несколько профилей. Концепция директора (director) отвечает за вызов операций, с помощью которых предоставляются ресурсы. Она работает с концепциями планировщика (planner) и движка (engine). Планировщик рассматривает существующий профиль, и определяет, какие операции нужно выполнить для того, чтобы преобразовать имеющийся вариант инсталляции в новое состояние. Движок отвечает за выполнение фактических операций предоставления ресурсов и за установку новых артефактов на диск. Концепция точекстыковки (touchpoints), являющаяся частью концепции движка, используется в тот момент, когда осуществляется установка. Например, для Eclipse SDK, есть точкистыковки Eclipse, зная которые можно устанавливать сборки. Для системы Linux, в которой Eclipse устанавливается из двоичных файлов RPM, движок должен использовать точкистыковки RPM. Кроме того, p2 может выполнять установку в параллель с другой работой или отдельно в отдельном процессе, таком как сборка приложения.

Новая система провизирования p2 имела много преимуществ. Артефакты, установленные в Eclipse, могли обновляться от выпуска к выпуску. Поскольку предыдущие профили хранились на диске, имелся также способ вернуться к предыдущему варианту инсталляции Eclipse. Кроме того, при наличии профиля и репозитария, вы могли воспроизвести вариант инсталляции Eclipse пользователя, который сообщил об ошибке, и попытаться воспроизвести проблему на своем собственном рабочем столе. Механизм предоставление ресурсов с помощью системы p2 давал больше, нежели просто возможность обновлять и устанавливать Eclipse SDK; это была платформа, которую можно было также применять и в случаях с RCP и OSGi. Команда разработчиков Equinox также работала с разработчиками другого проекта Eclipse, Eclipse Communication Framework (ECF), с тем, чтобы реализовать надежный транспорт, необходимый для артефактов и метаданных в репозитариях p2.

Когда в SDK была выпущена система p2, в сообществе Eclipse было много оживленных дискуссий. Поскольку менеджер обновлений был далеко не самым оптимальным решением в качестве системы провизирования конкретной инсталляции Eclipse, у пользователей Eclipse вошло в привычку распаковывать сборки в собственный вариант инсталляции и перезапускать Eclipse. Такой подход позволял надежным образом разрешать зависимости между сборками. Это также означало, что любые конфликты в вашей инсталляции разрешались во время выполнения, а не время установки. Ограничения следует учитывать во время установки, а не во время выполнения. Тем не менее, пользователи часто не обращали внимания на эти вопросы и поскольку сборки находились на диске, они считали, что сборки работали. Еще ранее варианты обновления, которые предлагал Eclipse, были простым каталогом, в котором находились сборки и возможности, представляющие собой архивы jar. В простом файле `site.xml` указывались имена возможностей, которые можно было использовать в обновленном варианте. С появлением p2, метаданные, которые были предоставлены в репозитариях p2, стали гораздо сложнее. Чтобы создать метаданные, процесс сборки необходимо было перенастроить так, чтобы либо генерировать метаданные во время сборки, либо запускать задачу генерации уже поверх уже существующих сборок. Первоначально не было доступной документации, в которой описывалось как делать эти изменения. А также, как это всегда бывает при предъявлении новой технологии более широкой аудитории, выявлялись неожиданные ошибки, которые следовало решать. Однако, когда было написано больше документации и были потрачены долгие часы на устранение всех этих ошибок, команда разработчиков Equinox смогла справиться со всеми этими проблемами и теперь система p2 стала базовым движком предоставления ресурсов, на котором основываются многие коммерческие предложения. Кроме того, фонд Eclipse Foundation каждый год предоставляет свой скоординированный релиз, используя для этого единый репозитарий p2 для всех проектов, в разработке которых он принимает участие.

6.4. Eclipse 4.0

Архитектуру нужно постоянно контролировать с тем, чтобы знать, является ли она по-прежнему актуальной. Есть ли возможность добавлять новые технологии? Должна ли она поощрять рост сообщества? Легко ли привлекать новых участников? В конце 2007 года участники проекта Eclipse решили, что ответов на эти вопросы не было и они приступили к разработке нового варианта видения проекта Eclipse. В то же самое время им стало понятно, что есть тысячи приложений Eclipse, которые зависят от существующего API. В конце 2008 года был создан технологический проект уровня инкубатора с целью решить следующие три конкретные задачи: сделать более простой модель программирования в Eclipse, привлечь новых участников проекта и позволить платформе воспользоваться новыми веб-технологиями, обеспечивая при этом открытость архитектуры.

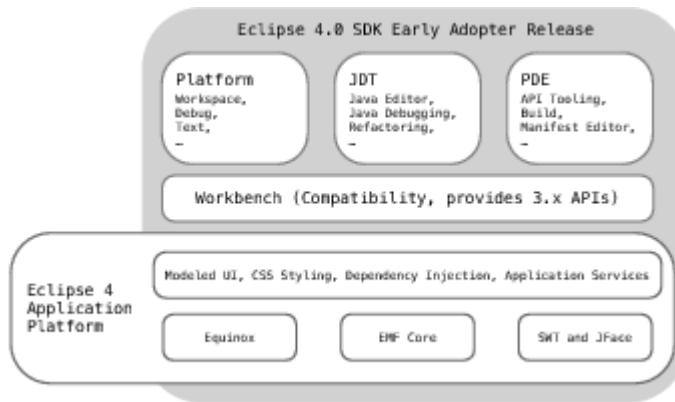


Рис.6.9: Ранний вариант релиза Eclipse 4.0 SDK

Первый пробный вариант релиза Eclipse 4.0 был в июле 2010 года с целью получить обратную связь от пользователей. Он состоял из комбинации сборок SDK, которые были частью релиза 3.6, а также новых сборок, которые были разработаны в рамках технологического проекта. Как и в случае релиза 3.0, был реализован слой совместимости, с тем чтобы существующие сборки могли работать с новым релизом. Как всегда, присутствовала оговорка, что для того, чтобы обеспечить эту совместимость, необходимо использовать общедоступный интерфейс API. Такой гарантии не было, если в вашей сборке использовался внутренний код предыдущих релизов. В релизе 4.0 была предложена платформа приложений Eclipse 4 Application Platform, обладающая следующими возможностями.

6.4.1. Рабочее пространство модели

В версии 4.0 рабочее пространство модели создается при помощи фреймворка Eclipse Modeling Framework (EMFgc). Есть различие между моделью (model) и внешним видом (view), т.к. механизм, осуществляющий визуализацию, опрашивает модель, а затем генерирует код SWT. По умолчанию используются средства визуализации SWT, но допустимы другие решения. Если вы создаете пример приложения 4.x, то для модели рабочего пространства, используемого по умолчанию, будет создан файл XMI. Модель можно изменять и рабочее пространство будет мгновенно обновляться в соответствие с изменениями в модели. На рис.6.10 приведен пример модели, созданной для примера приложения 4.x.

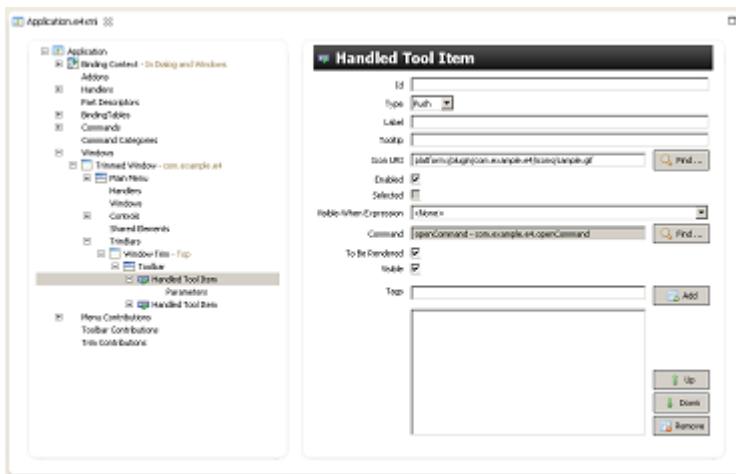


Рис.6.10: Модель, созданная для примера приложения 4.x

6.4.2. Стиль каскадных стилевых страниц

Eclipse был выпущен в 2001 году, еще до эпохи появления многофункциональных интернет-приложений, внешний вид которых можно изменяться с помощью CSS. В Eclipse 4.0 есть возмож-

ность использовать стили для более простого изменения внешнего вида приложения Eclipse. По умолчанию таблицы стилей CSS можно найти в каталоге `css` в сборке `org.eclipse.platform`.

6.4.3. Внедрение зависимостей

Примерами моделей программирования сервисов являются реестр расширений Eclipse и сервисы OSGi. В соответствие с соглашением, в модели программирования сервисов имеются поставщики и потребители сервисов. За налаживание отношений между поставщиками и потребителями несет ответственность брокер.

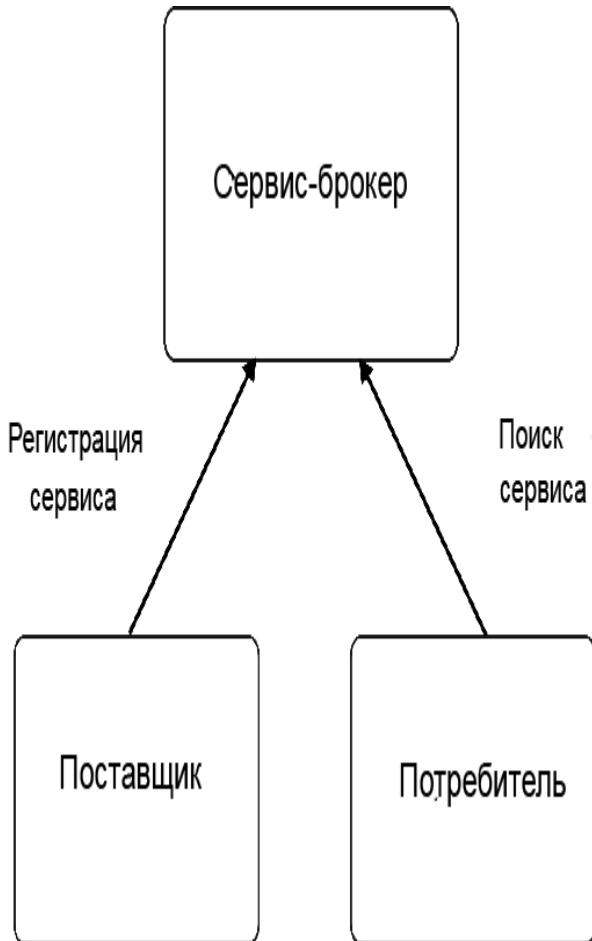


Рис.6.11: Взаимосвязь между поставщиками и потребителями

В приложениях Eclipse 3.4.x потребителю для того, чтобы использовать сервисы, обычно необходимо знать расположение реализаций сервисов, а также понимать порядок наследования во фреймворке. Поэтому код, который создает потребитель, становится менее пригодным для повторного использования, т. к. те, кто им пользуется, не могут переопределить реализацию, которая определена потребителем. Например, если вы хотите обновить сообщение в строке состояния в Eclipse 3.x, то код будет выглядеть следующим образом:

```
getViewSite().getActionBars().getStatusLineManager().setMessage(msg);
```

Eclipse, 3.6 построен из компонентов, но многие из этих компонентов также слишком сильно взаимозависимы. Для того, чтобы можно было собирать приложения из менее связанных между собой компонентов, в Eclipse 4.0 для предоставления сервисов клиентам используется внедрение зависимостей (dependency injection). Внедрение зависимостей Eclipse, 4.x осуществляется через специальный фреймворк, в котором используется концепция контекста, служащая в качестве общего механизма поиска сервисов потребителями. Контекст существует между приложением и фреймворком. Контексты являются иерархическими. Если в контекст поступил запрос, который не

удается разрешить, то произойдет делегирование запроса в родительский контекст. В контексте Eclipse, который называется `IEclipseContext`, хранятся имеющиеся сервисы и происходит поиск сервисов OSGi. По сути контекст похож на то, как в языке Java происходит отображение имен или классов в объекты. Контекст осуществляет обработку элементов модели и ее сервисов. Каждый элемент модели будет иметь контекст. Сервисы публикуются в версиях 4.x при помощи механизма сервисов OSGi.

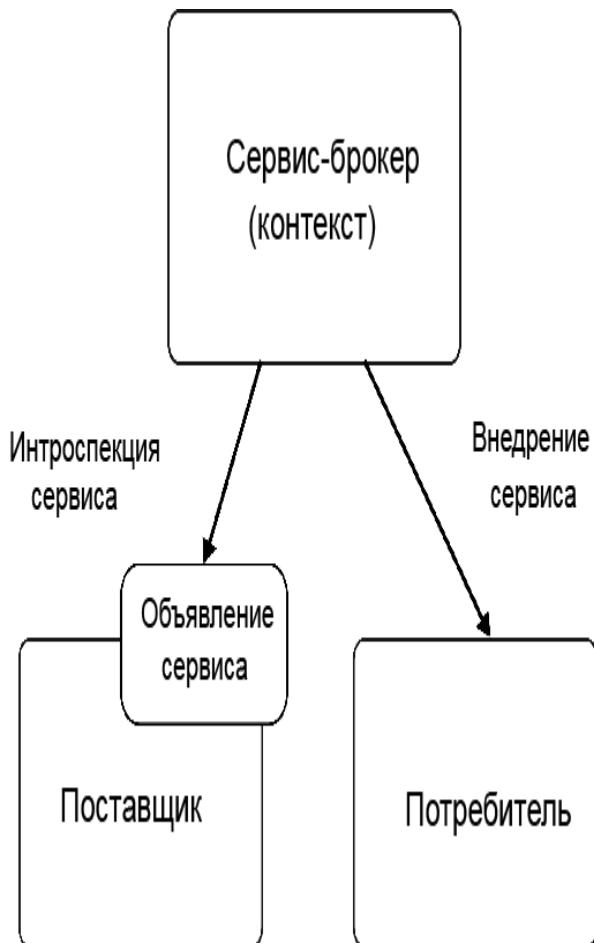


Рис.6.12: Контекст брокера сервисов

Поставщики добавляют в контекст сервисы и объекты, которые там хранятся. Сервисы внедряются в объекты потребителей при помощи контекста. Потребитель объявляет, что ему нужно, и контекст определяет, как удовлетворить этот запрос. Такой подход упростил использование динамических сервисов. В Eclipse 3.x, потребителю нужно было подключать слушателей (listeners) для того, чтобы получать уведомления о том, когда сервисы становятся доступными или недоступными. В Eclipse 4.x как только контекст внедряется в объект потребителя, любые изменения будут автоматически снова поступать в этот объект. Иными словами, снова будет происходить внедрение зависимостей. Потребитель указывает, что он будет использовать контекст, с помощью аннотаций Java 5, которые соответствуют стандарту JSR 330, а именно с помощью `@inject`, а также с помощью некоторых специально настроенных аннотаций проекта Eclipse. Поддерживается внедрение в конструкторы, методы и поля. Среда выполнения релизов 4.x ищет в объектах такие аннотации. Выполняемое действие зависит от того, какая была найдена аннотация.

Такое разделение контекста и приложения позволяет улучшить возможность повторного использования компонентов, и освобождает потребителя от необходимости разбираться в их реализации. В релизе 4.x код, обновляющий строку состояния, будет выглядеть следующим образом:

```

@Inject
IStatusLineManager statusLine;
:   :   :

```

```
statusLine.setMessage(msg);
```

6.4.4. Сервисы приложений

Одной из главных задач в Eclipse 4.0 было создание настолько простого интерфейса API, что с его помощью было легко реализовывать сервисы общего назначения. Был создан список простых сервисов, считающихся «наиболее важными» и известными как сервисы приложений Eclipse Application. Задача состоит в предоставлении автономно работающего интерфейса API, которыми клиенты могут пользоваться без необходимости глубокого понимания всех доступных интерфейсов API. Сервисы приложений были реализованы в виде индивидуально работающих сервисов, так что ими также можно пользоваться в других языках, отличающихся от языка Java, таких как Javascript. Например, есть интерфейс API для доступа к модели приложения, который позволяет читать и изменять настройки приложения (preferences) и выдавать сообщения об ошибках и предупреждения.

6.5. Заключение

Архитектура Eclipse, базирующаяся на компонентах, разрабатывалась таким образом, чтобы можно было добавлять новые технологии и, при этом, сохранять обратную совместимость. На это потребовались затраты, но наградой был рост сообщества Eclipse, т. к. потребители были уверены, что они могут продолжать поставлять свои разработки, базирующиеся на стабильном API.

Eclipse используется настолько широко, причем варианты применения нашего обширного API настолько разнообразны, что новичкам становится трудно его адаптировать и понимать. Если оглянуться назад, то мы бы должны были делать так, чтобы наш интерфейс API оставался простым. Если 80% потребителей используют только 20% возможностей интерфейса API, то это означает, что есть необходимость сделать его более простым, и это одна из причин, почему была создана ветка Eclipse 4.x.

Мудрость толпы действительно подсказывает интересные случаи применения, например, деление IDE на сборки, которые могут использоваться для создания приложений RCP. С другой стороны, толпа часто создает много шума из-за просьб о решениях, относящихся к крайним случаям, на реализацию которых требуется значительное время.

В первые дни работы над проектом Eclipse, для разработчиков было роскошью тратить значительное количество времени на документацию, примеры и ответы на вопросы сообщества. Со временем, эта обязанность в целом перешла к сообществу Eclipse. Для того, чтобы помочь сообществу, мы могли бы написать лучшую документацию и привести лучшие варианты применения, но тогда было бы сложно реализовывать большое количество изменений и добавлений, которые планируются для каждого выпуска. Вопреки ожиданиям, что даты выпуска программ могут быть сдвинуты, мы продолжаем публиковать выпуски Eclipse во-время, что позволяет нашим пользователям надеяться, что они могут делать то же самое.

Мы адаптируем новые технологии, изобретаем, как будет выглядеть и работать Eclipse, и мы продолжаем общение с нашими пользователями, поддерживая их сообщество. Если вы хотите принять участие в проекте Eclipse, пожалуйста, посетите сайт <http://www.eclipse.org>.

Примечания

1. <http://www.eclipse.org>
2. <http://www.eclipse.org/equinox>
3. Например: <http://help.eclipse.org>.

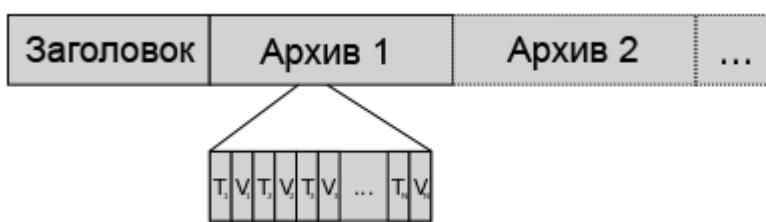
7.Проект Graphite

Graphite [1] выполняет две довольно простые задачи: хранение данных, изменяемых со временем, и отображение их в виде графиков. В течение многих лет для выполнение подобных задач было написано большое количество программ. Уникальность проекта Graphite состоит в том, что он предоставляет эти функции в виде сервиса, который прост в использовании и хорошо масштабируем. Протокол ввода данных в Graphite настолько прост, что вы можете научиться делать это самостоятельно за несколько минут (не то, чего вам бы, на самом деле, хотелось, но это достойная лакмусовая бумажка для проверки его простоты). Отображение графиков и поиск точек, соответствующих определенным данным, столь же прост, как заполнение адреса URL. Это позволяет вполне естественно интегрировать проект Graphite с другим программным обеспечением и предоставляет пользователям возможность собирать мощные приложения на базе Graphite. Одним из наиболее распространенных применений пакета Graphite является создание веб-панелей управления, используемых для мониторинга и анализа данных. Пакет Graphite появился в среде крупномасштабной электронной коммерции, что отразилось в его архитектуре. Его ключевыми особенностями являются масштабируемость и доступ к данным в режиме реального времени.

К числу компонентов, которые позволяют проекту Graphite реализовывать эти цели, относятся специализированная библиотека баз данных со своим собственным форматом хранения данных, механизм кеширования для оптимизации операций ввода/вывода и простой, но эффективный метод кластеризации серверов Graphite. Вместо того, чтобы просто описывать, как Graphite работает в настоящее время, я объясню, как первоначально (достаточно наивно) был реализован Graphite, с какими проблемами я столкнулся и какие были использованы пути их решения.

7.1. Библиотека базы данных: хранение данных временных рядов

Graphite был написан на языке Python и состоит из трёх основных компонентов: библиотеки баз данных `whisper`, демона серверной стороны `carbon`, и клиентского веб-приложения, визуализирующего графики и предоставляющее базовый пользовательский интерфейс. Несмотря на то, что библиотека `whisper` была разработана специально для системы Graphite, она может использоваться независимо от этой системы. По своей конструкции она очень похожа на кольцевую базу данных, используемую в `RRDtool`, и хранит только числовые данные временных рядов. Обычно мы считаем, что базы данных являются серверными процессами, с которыми клиентские приложения общаются через сокеты. Однако, `whisper` точно также, как и `RRDtool` является библиотекой баз данных, используемой приложениями для обработки и поиска данных, хранящихся в файлах, оформленных специальным образом. Самыми главными операциями библиотеки `whisper` являются операция `create`, создающая новый файл `whisper`, операция `update`, записывающая новые данные в файл, и операция `fetch`, используемая для выборки данных.



(`timestamp, value`) (отметка о времени, значение). Когда выполняется операция обновления `update` или выборки `fetch, whisper` по отметке о времени и конфигурации архива определяет смещение в файле, в который должна быть сделана запись или из которого должно быть выполнено чтение.

7.2. Серверная часть: Простой сервис хранения данных

Серверная часть проекта Graphite представляет собой процесс-демон, называемый `carbon-cache`, на который обычно ссылаются как на `carbon`. Он собран на Twisted, хорошо масштабируемом управляемым событиями фреймворке ввода/вывода для языка Python. Twisted позволяет серверу `carbon` эффективно общаться с большим количеством клиентов и обрабатывать большой объём трафика с малыми накладными расходами. На рис.7.2 показан поток данных, идущий между `carbon`, `whisper` и веб-приложением: клиентские приложения собирают данные и отсылают их на серверную сторону Graphite в `carbon`, где данные хранятся в файлах `whisper`. Затем эти данные могут использоваться веб-приложением Graphite для создания графиков.

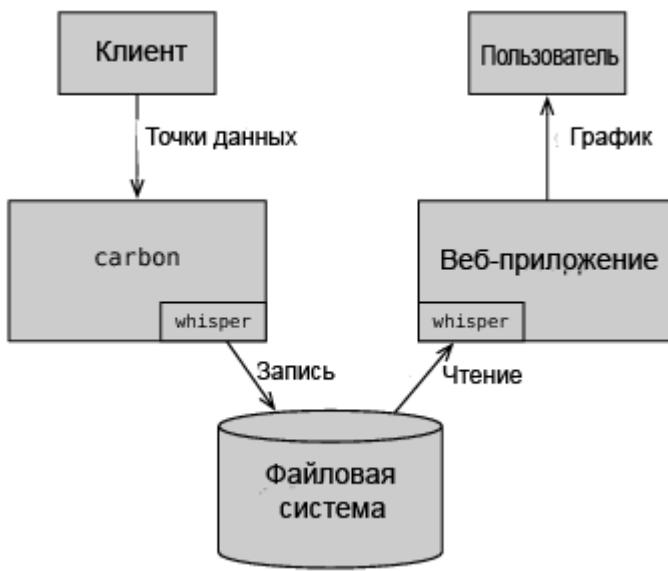


Рис.7.2: Поток данных

Основная функция демона `carbon` состоит в запоминании значений точек данных для метрик, предоставляемых клиентами. В терминологии Graphite метрикой является любая сущность, которую можно измерить и которая изменяется с течением времени (например, использование ресурсов процессора сервера или объем продаж какого-либо продукта). Точка данных является просто парой (`timestamp, value`), соответствующей значению конкретной метрики, измеренному в определенное время. Метрики идентифицируются уникальным образом с помощью собственного имени, причем имена каждой метрики точно также, как и точки данных передаются из клиентских приложений. Обычным типом клиентского приложения является агент, осуществляющий мониторинг, который собирает значения системных или прикладных метрик и отсылает собранные значения демону `carbon` для хранения и визуализации. Метрики в Graphite имеют простые, иерархические имена, похожие на пути в файловых системах и отличающиеся лишь тем, что в качестве разделителя иерархии используется точка, а не слэш или обратный слэш. `Carbon` берет любое допустимое имя и для каждой метрики создает файл `whisper`, в котором хранятся точки данных этой метрики. Файлы `\code{whisper}` хранятся в каталоге данных `\code{carbon}` в виде иерархической файловой системы, отражающей иерархию деления точками на части имени каждой метрики. Следовательно `servers.www01.cpuUsage` отображается, например, в `.../servers/www01/cpuUsage.wsp`.

Когда клиентское приложение хочет отправить в Graphite значения точек данных, оно должно установить с `carbon` обычно через порт 2003 [2] соединение TCP. Все сообщения поступают только

со стороны клиента: carbon ничего не пересыпает через это соединение. Пока соединение остается открытым, клиент, когда это необходимо, отсылает значения точек данных в простом текстовом формате. Формат представляет собой одну текстовую строку для каждого значения точки ввода, в которой указываются разделяемые пробелами имя, имеющее точки, значение данных и временная отметка, использующая формат времени UNIX. Например, клиент может послать следующее:

```
servers.www01.cpuUsage 42 1286269200
products.snake-oil.salesPerMinute 123 1286269200
[one minute passes]
servers.www01.cpuUsageUser 44 1286269260
products.snake-oil.salesPerMinute 119 1286269260
```

Если рассматривать в общем, то все, что carbon делает, это прослушивает поступающие данные в этом формате и с помощью whisper пытается сохранить их на диске настолько быстро, насколько это возможно. Далее мы обсудим некоторые конкретные особенности, используемые для обеспечения масштабируемости и достижения наилучшей производительности, которую мы можем получить на обычном жёстком диске.

7.3. Клиентская часть: Графики по запросу

Веб-приложение Graphite позволяет пользователям запрашивать различные графики с помощью простого интерфейса API, базирующегося на использовании URL. Параметры графика указываются в строке HTTP-запроса GET, а в ответ возвращается изображение в формате PNG. Например, с помощью URL

```
http://graphite.example.com/render?target=servers.www01.cpuUsage&
width=500&height=300&from=-24h
```

делается запрос графика размером 500×300 для метрики servers.www01.cpuUsage с данными за последние 24 часа. В действительности нужно указать только целевой параметр (т. е. только то, что надо отобразить — *прим.пер.*); остальные параметры являются необязательными и, если их не указывать, то будут использоваться значения, задаваемые по умолчанию.

В проекте Graphite поддерживается большое количество разных параметров выдачи изображений, а также функций обработки данных, которые можно указывать с помощью простых функционально понятных синтаксических правил. Например, нам нужен график скользящих средних значений, рассчитываемых по десяти точкам, для метрики, взятой из предыдущего примера:

```
target=movingAverage(servers.www01.cpuUsage,10)
```

Функции могут быть вложенными, что позволяет создавать составные выражения и выполнять сложные вычисления.

Ниже показан еще один пример, в котором приводится промежуточная сумма с нарастающим итогом продаж за день, изображающая поминутное значение метрик для каждого продаваемого продукта:

```
target=integral(sumSeries(products.*.salesPerMinute))&from=midnight
```

С помощью функции sumSeries вычисляется временной ряд, являющийся суммой значений каждой метрики, соответствующей образцу products.*.salesPerMinute. Затем с помощью функции integral рассчитывается промежуточная сумму с нарастающим итогом с интервалом в одну минуту. Из этих примеров несложно понять, как можно создать пользовательский веб-интерфейс для просмотра и обработки графиков. Пакет Graphite поставляется с своим собственным конструктором пользовательского интерфейса Composer UI, показанным на рис.7.3, в котором, когда пользо-

ватель выбирает из меню имеющиеся возможности, то используется язык JavaScript, который изменяет параметры URL запрашиваемого графика.

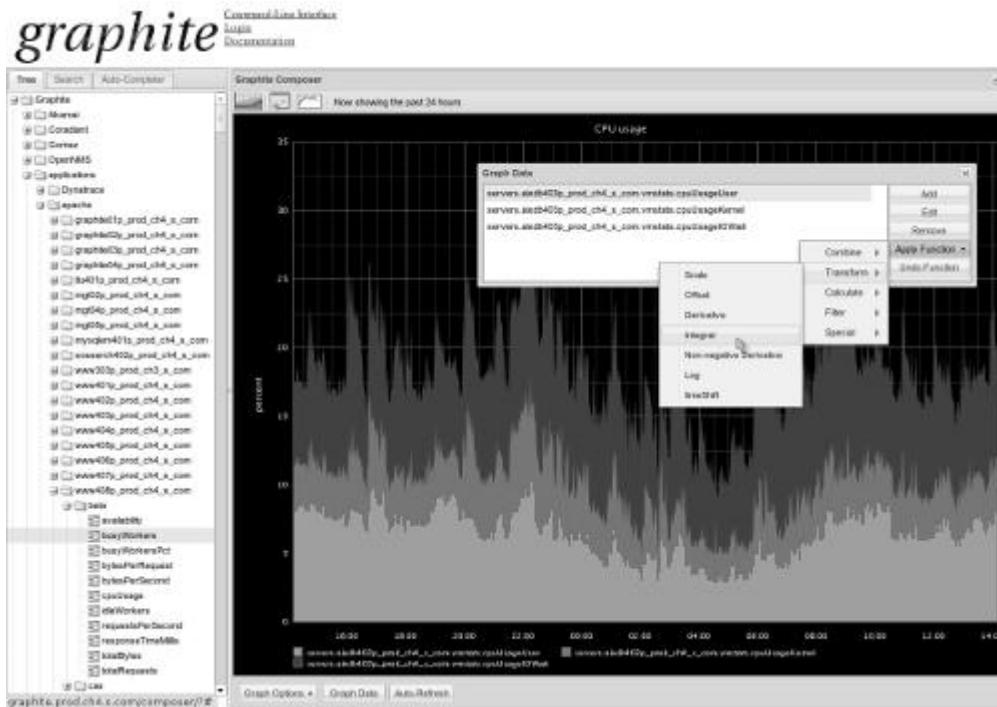


Рис.7.3: Внешний вид конструктора интерфейсов пакета Graphite

7.4. Панели управления

С самого своего появления пакет Graphite использовался как инструмент для создания панелей управления на основе веб-интерфейса. Интерфейс URL API делает такое использование вполне естественным. Создать панель управления столь же просто, как добавить теги в HTML-страницу при ее создании, например:

```

```

Однако, не всем нравится вручную прописывать адреса URL, поэтому в конструкторе пользовательских интерфейсов пакета Graphite для создания графиков предлагается метод «выбери точку и щелкни», с помощью которого вы можете просто скопировать и вставить адрес URL. Если объединить этот метод с еще одним инструментом для быстрого создания веб-страниц (например, страниц wiki), то это позволит даже не сильно подкованным в техническом смысле пользователям сравнительно легко собирать свои собственные панели управления.

7.5. Очевидное узкое место

Как только мои пользователи начали создавать панели управления, у Graphite быстро появились проблемы с производительностью. Я проанализировал журналы веб-сервера с тем, чтобы увидеть, какие запросы тянут систему на дно. Абсолютно очевидно, что проблема была в запросе огромного количества графиков. Веб-приложение сильно загружало центральный процессор, постоянно рисуя графики. Я заметил, что было масса идентичных запросов и виной тому были панели управления.

Представьте себе, что у вас есть панель управления с десятью графиками, и страница обновляется каждую минуту. Каждый раз, когда пользователь открывает в браузере панель управления, Graphite должен обработать за минуту еще на 10 запросов больше. Это быстро становится затратным.

Простым решением было рисовать каждый график только один раз, а затем каждому пользователю отправлять его копию. Веб-фреймворк Django (на котором был построен проект Graphite) предоставляет замечательный механизм кэширования, в котором можно использовать различные серверные решения, например, memcached. Memcached [3] является, в сущности, хеш-таблицей, предоставляемой в виде сетевого сервиса. Клиентские приложения могут получать и устанавливать пары «ключ-значение» точно также, как и в обычной хеш-таблице. Главное преимущество в использовании memcached состоит в том, что результат затратного запроса (например, на визуализацию графика) может быть очень быстро запомнен и позднее может быть найден при обработке последующих запросов. Чтобы навсегда избежать ситуаций с возвратом устаревших графиков, memcached можно сконфигурировать таким образом, чтобы время хранения кэша истекало в течение очень короткого периода времени. Даже если это будет всего лишь несколько секунд, для Graphite это огромное облегчение, поскольку дублирующие запросы распространены очень сильно.

Другим типичным случаем, при котором создается большое количество запросов на визуализацию, является ситуация, когда пользователь настраивает параметры отображения и использует функции в конструкторе пользовательского интерфейса. Каждый раз, когда пользователь что-то изменяет, Graphite должен перерисовать график. В каждом запросе указываются одни и те же данные, поэтому имеет смысл в memcached также поместить данные, которые использовались для визуализации. Это позволяет пользовательскому интерфейсу продолжать реагировать на запросы пользователей, поскольку этап выборки данных пропускается.

7.6. Оптимизация ввода/вывода

Представьте себе, что у вас есть 60000 метрик, которые вы отправляете на ваш сервер Graphite, и для каждой из этих метрик есть одно значение точки данных в течение одной минуты. Вспомните, у каждой метрики в файловой системе есть свой собственный файл whisper. Это означает, что carbon должен каждую минуту выполнять по одной операции записи в 60000 различных файлов. Пока carbon может делать запись в один файл за миллисекунду, он будет справляться с ситуацией. Следующая ситуация отстоит от текущей не так уж далеко, но давайте предположим, что каждую минуту вы должны обновлять 600000 метрик, или ваши метрики обновляются каждую секунду, или, возможно, вы просто не можете себе позволить использовать достаточно быструю память. Независимо от ситуации, предположим, скорость поступающих значений точек данных превышает скорость операций записи, которую может поддерживать ваша память. Как справиться с такой ситуацией?

Большинство современных жёстких дисков имеет медленное время позиционирования [4], т.е. имеет большую задержку между выполнением операций ввода/вывода в двух различных местах на диске в сравнении с записью непрерывной последовательности данных. Это означает, что, чем больше мы делаем непрерывных записей, тем большую производительность мы получаем. Но если у нас тысячи файлов, в которые необходимо часто делать запись, и каждая запись очень мала (одно значение точки данных в whisper занимает всего 12 байтов), то жёсткие диски, несомненно, будут тратить большую часть времени на позиционирование.

Если исходить из условия, что скорость выполнения операций записи имеет сравнительно низкую планку, единственный способ сделать так, чтобы пропускная способность наших значений точек данных превышала эту планку, это записывать значения нескольких точек данных в одной операции записи. Это возможно, поскольку whisper записывает подряд идущие значения точек данных на диск непрерывно друг за другом. Поэтому я добавил в whisper функцию update_many, которая берет список значений точек данных для одной метрики и собирает значения подряд идущих точек данных в одну операцию записи. Даже хотя размер каждой записи становится больше, разница во времени записи десяти значений точек данных (120 байтов) в сравнении с записью значения одной точки данных (12 байтов) незначительна. Можно взять достаточно много значений точек данных прежде, чем размер каждой записи начнет вносить существенную задержку.

Затем я реализовал в carbon механизм буферизации данных. Каждое поступающее значение точки данных отображается в очереди, имеющей имя метрики, и, затем, помещается в эту очередь. Еще один поток циклически проходит по всем очередям и для каждой из них выбирает все значения точек данных и с помощью функции `update_many` записывает их в соответствующий файл `whisper`. Если вернуться к приведённому выше примеру, когда у нас есть 600000 метрик, обновляющихся каждую минуту, и наша память может справиться только с одной записью в миллисекунду, то в среднем в каждой очереди может быть приблизительно до 10 значений точек данных. Единственным ресурсом, которым мы за это платим, является память, которой хватает с избытком поскольку каждая точка данных занимает всего несколько байтов.

Эта стратегия позволяет динамически накапливать в буфере столько точек данных, сколько необходимо для поддержки скорости поступающих точек данных, которая может превышать скорость операций ввода/вывода и с которой может справиться ваша память. Хорошим преимуществом такого подхода является то, что он повышает степень устойчивости к временным замедлениям ввода/вывода. Если системе нужно выполнить другую работу по вводу/выводу, не связанную с Graphite, то, скорее всего, скорость операций записи уменьшится, что просто приведет к росту очередей в carbon. Чем больше очереди, тем больше размер записи. Поскольку общая пропускная способность значений точек данных равна скорости операций записи, умноженной на средний размер каждой записи, carbon способен держаться до тех пор, пока для очередей будет достаточно памяти. Механизм очередей в carbon изображён на рис.7.4.

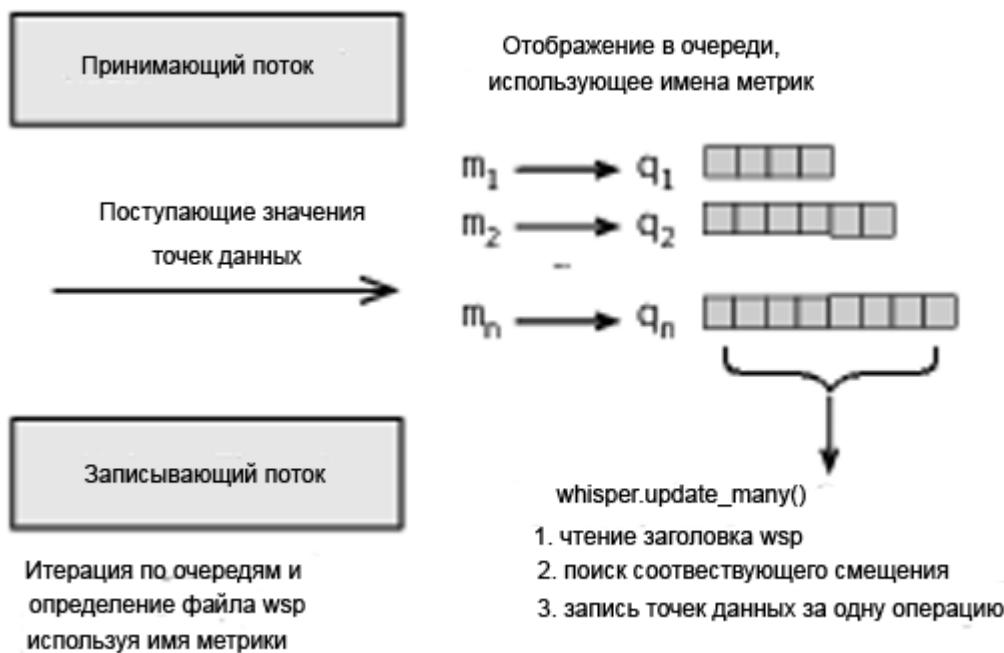


Рис.7.4: Механизм очередей в Carbon

7.7. Все это в режиме реального времени

Буферизация значений точек данных была замечательным способом оптимизировать ввод/вывод в carbon, но через не очень продолжительное время мои пользователи заметили довольно тревожный побочный эффект. Снова вернемся к примеру с 600000 метриками, обновляющимися каждую минуту, и, предположим, что наша память может справиться лишь с 60000 операциями записи в минуту. Это означает, что у нас возникает ситуация, когда в любой заданный момент времени данные будут находиться в очередях carbon приблизительно по 10 минут. Для пользователя это означает, что в графиках, которые они запросили из веб-приложения Graphite, будут отсутствовать данные за последние 10 минут. Это плохо!

К счастью, решение является довольно простым. Я добавил в `carbon` слушающий сокет, в котором предлагается интерфейс для работы с очередью, предоставляющий доступ к точкам данным, запомненным в буфере, а затем изменил веб-приложение `Graphite` так, чтобы использовать этот интерфейс каждый раз, когда нужно найти данные. Затем веб-приложение комбинирует значения точек данных, полученных из `carbon`, со значениями точек данных, считываемых с диска и, вуала, графики строятся в режиме реального времени. Конечно, в нашем примере значения точек данных обновляются раз в минуту и, следовательно, не совсем «в реальном времени», но тот факт, что каждое значение точки данных мгновенно появляется на графике сразу, как только оно поступило в `carbon`, считается режимом реального времени.

7.8. Ядра, кэширование и катастрофические отказы

К настоящему моменту, вероятно, уже очевидно, что ключевой характеристикой производительности, от которой зависит собственная производительность `Graphite`, является задержка ввода/вывода. До сих пор мы предполагали, что в нашей системе стабильно низкая задержка ввода/вывода, составляющая в среднем около одной миллисекунды на запись, но это серьезное предположение, требующее несколько более глубокого анализа. Большинство жестких дисков просто не настолько быстры; даже когда десятки дисков объединены в RAID-массив, есть очень большая вероятность, что задержка при случайном доступе будет больше одной миллисекунды. Тем не менее, если вы попробуете и проверите, насколько быстро даже старый ноутбук мог бы записать целый килобайт данных на диск, вы обнаружите, что системный вызов записи возвращает управление гораздо быстрее, чем 1 миллисекунда. Почему?

Всякий раз, когда характеристики производительности программ оказываются противоречивыми или неожиданными, в этом виноваты, как правило, либо буферизация, либо кэширование. В данном случае мы имеем дело с обеими причинами. Системный вызов записи технически не выполняет запись ваших данных на диск, он просто помещает данные в буфер, который позже ядро запишет на диск. Поэтому вызов записи обычно возвращается так быстро. Даже после того, как буфер был записан на диск, он часто остается в кеш-памяти для последующих операций чтения. Для обеих этих функций, буферизации и кэширования, конечно, требуется память.

Разработчики ядра, будучи умные люди, решили, что будет хорошо, если использовать любую свободную память пользователя пространства, а не выделять память напрямую. Это оказывается чрезвычайно полезным способом повышения производительности и это также объясняет, почему независимо от того, сколько к системе вы добавляете памяти, количество «свободной» памяти после выполнения небольшого количества операций ввода/вывода в конечном итоге стремится к нулю. Если ваши приложения, размещенные в пользовательском пространстве, не используют эту память, то ядро, вероятно, использует. Недостатком этого подхода является то, что эта «свободная» память может быть забрана из ядра в тот момент, когда приложение пользовательского пространства решит, что ему для собственных нужд требуется выделить больше памяти. У ядра нет выбора, кроме как отказаться от памяти, потеряв все, возможно, и буфера, которые там были.

Что это всё означает для пакета `Graphite`? Мы просто подчеркнули зависимость `carbon` от стабильно низкой задержки ввода/вывода, и мы также знаем, что системный вызов записи возвращает управление быстро только потому, что данные просто копируются в буфер. Что происходит, когда ядру не хватает памяти для продолжения буферизации записи? Запись становится синхронной и, следовательно, ужасно медленной! Это приводит к резкому снижению скорости операций записи в `carbon`, что ведет в `carbon` к росту очередей, съедающих еще больше памяти, которой еще больше не хватает ядру. В конце концов, такая ситуация обычно приводит к тому, что в `carbon` заканчивается память или что сердитый сисадмин убивает процесс.

Чтобы избежать катастрофы подобного рода, я добавил к `carbon` несколько возможностей, в том числе конфигурируемые ограничения на количество значений точек данных, которые могут быть в очереди, и ограничения, определяющие насколько быстро в `{whisper}` могут выполняться раз-

личные операции. Эти возможности могут защитить carbon от потери управления, но вместо этого могут привести к менее суровым последствиям, например, к пропаданию некоторых значений точек данных и к отказу от приема новых значений точек данных. Однако, правильные значения для этих настроек зависят от системы и для их настройки требуется значительное количество проб. Эти возможности полезны, но они не решают данную проблему принципиально. Для этого нам понадобится больше оборудования.

7.9. Кластеризация

Объединить несколько серверов Graphite так, чтобы они казались одной системой, с точки зрения пользователя не так уж и трудно, по крайней мере, при наивной реализации. Взаимодействие веб-приложения с пользователем в первую очередь состоит из двух операций: поиск метрик и выборка значений для точек данных (как правило, в виде графика). Операции поиска и выборки из веб-приложения спрятаны в библиотеке, которая абстрагирует их реализацию от остальной части кода, причем они также доступны через обработчики запросов HTTP запроса для того, чтобы их было проще вызывать дистанционно.

Операция поиска `find` ищет в локальной файловой системе `whisper` совпадение с образцом, заданным пользователем точно также, как общая файловая система ищет совпадение файлов, например, `*.txt`, с указанным расширением. Поскольку это древовидная структура, результат,озвращаемый операцией `find`, является коллекцией объектов `Node` (Узел), каждый из которых является производным подкласса `Branch` (Ветка) или `Leaf` (Лист) класса `Node`. Каталоги соответствуют узлам-веткам, а файлы `whisper` - узлам-листьям. Такой уровень абстракции упрощает поддержку хранилищ данных, расположенных ниже, в том числе файлов RRD [5] и заархивированных файлов `\code{whisper}`.

В интерфейсе `leaf` определяется метод выборки `fetch`, реализация которого зависит от типа листа. В случае, если это файл `whisper`, он является просто тонкой оболочкой вокруг собственной функции `fetch` библиотеки `whisper`. Когда была добавлена поддержка кластеризации, функция `find` была расширена так, чтобы она могла выполнять дистанционные вызовы функции `find` через протокол HTTP к другим серверам Graphite, указанным в конфигурации веб-приложения. Данные об узлах, содержащиеся в результатах этих HTTP-запросов, предоставляются в виде объектов `RemoteNode`, которые пригодны для использования с интерфейсами `Node`, `Branch` и `Leaf`. Это делает кластеризацию прозрачной для остальной части кода веб-приложения. Метод `fetch` для удалённых узлов-листьев выполняется как еще один запрос HTTP для получения значений точек данных с сервера Graphite, где расположен этот узел.

Все эти вызовы выполняются между веб-приложениями точно также, как они выполняются клиентом, за исключением лишь одного параметра, указывающего, что операция должна выполняться локально, а не перераспределяться по кластеру. Когда в веб-приложение приходит запрос на визуализацию графика, оно выполняет операцию `find` с тем, чтобы найти запрашиваемые метрики, и вызывает операцию `fetch` для каждой из них для того, чтобы выбрать значения их точек данных. Это работает во всех случаях: когда данные находятся на локальном сервере, на удалённом сервере, или на локальном и удаленном серверах одновременно. Если сервер выходит из строя, то сравнительно быстро возникает состояние таймаута для дистанционных вызовов и сервер помечается как необслуживаемый на короткий промежуток времени, в течение которого к нему не будет запросов. С точки зрения пользователя, все данные, которые были на потерянном сервере, будут отсутствовать на их графиках, если, конечно, эти данные не были продублированы на другом сервере в кластере.

7.9.1. Краткий анализ эффективности кластеризации

Самой дорогостоящей частью запроса графика является отрисовка графика. Каждая отрисовка выполняется одним сервером, поэтому добавление большего количества серверов эффективно уве-

личивает производительность отрисовки графиков. Однако тот факт, что многие запросы перераспределяют вызовы `find` на любой другой сервер в кластере, означает, что наша схема кластеризации используется в перераспределении нагрузки, касающейся клиентских запросов, а не рассредоточения нагрузок. Однако, то, чего мы добились на данный момент, является эффективным способом распределения серверных нагрузок, поскольку каждый экземпляр `carbon` работает независимо от других. Это хороший первый шаг, поскольку в большинстве случаев серверная сторона становится узким местом задолго до того, как это происходит с клиентскими обращениями, но очевидно, что с помощью этого подхода не удастся выполнить горизонтальное масштабирование клиентских обращений.

Что сделать масштабирование клиентских обращений более эффективным, необходимо сократить количество дистанционных запросов `find`, делаемых веб-приложением. Снова простейшим решением является кеширование. Точно также, как `memcached` уже используется для кеширования значений точек данных и для нарисованных графиков, этот же самый подход можно использовать для кеширования результатов запросов `find`. Поскольку маловероятно, что местоположение метрик меняется часто, их можно хранить в кэше достаточно долго. Однако если настройка таймаута кеширования результатов операций `find` окажется слишком длительной, то новые метрики, которые добавлены в иерархию, могут не сразу стать доступными для пользователя.

7.9.2. Хранение метрик в кластере

В кластере веб-приложение Graphite сравнительно однородное в том смысле, что оно выполняет на каждом сервере одну и ту же работу. Однако, роль `carbon` может варьироваться от сервера к серверу в зависимости от того, какие данные отправляются на каждый экземпляр сервера. Часто есть много различных клиентов, посылающих данные в `carbon`, поэтому было бы весьма хлопотно настраивать конфигурацию каждого клиента в соответствие с компоновкой кластера Graphite. Метрики, используемые в обычных приложениях, могут отсылаться на один сервер `carbon`, а метрики, используемые в бизнес-приложениях, - на несколько серверов `carbon` с целью обеспечения избыточности.

Чтобы упростить управление сценариями подобного рода, пакет Graphite поставляется с дополнительным инструментальным средством, называемым `carbon-relay`. Его работа сравнительно проста; он получает от клиентов данные метрики точно также, как и стандартный демон `carbon` (а точнее, `carbon-cache`), но вместо того, чтобы запоминать данные, он применяет к именам метрик набор правил с тем, чтобы определить, на какой сервер `carbon-cache` перенаправить данные. Каждое правило состоит из регулярного выражения и списка серверов, на которые можно направлять данные. Для каждого значения точки данных по порядку применяются правила и используется первое правило, регулярное выражение в котором совпадет с именем метрики. Таким образом всё, что нужно клиенту сделать, это отослать данные на `carbon-relay` и попадут на надлежащие серверы.

В каком-то смысле `carbon-relay` предоставляет функциональные возможности репликации, хотя более точно они должны называться дублированием входных данных, поскольку не решаются вопросы синхронизации. Если сервер временно недоступен, то будут отсутствовать значения точек данных за тот период, в течение которого он был недоступен, но все остальное будет функционировать нормально. Есть административные скрипты, которые позволяют системному администратору управлять процессом ресинхронизации.

7.10. Размышления о проекте

Мой опыт работы с пакетом Graphite подтвердил мое убеждение, что масштабируемость имеет очень мало общего с низким уровнем производительности, а является результатом общего устройства проекта. По пути я сталкивался со многими узкими местами, но каждый раз я искал решения в преобразовании проекта, а не в повышении производительности. Я много раз спрашивал, почему

я написал Graphite на языке Python, а не на Java или на C++, и я всегда отвечал, что я до сих пор не гонюсь за той производительностью, что мог бы предложить другой язык. В работе [Knu74], Дональд Кнут сказал знаменитую фразу, что преждевременная оптимизация есть корень всех зол. Пока мы считаем, что наш код будет продолжать развиваться нетривиальными способами, любая оптимизация [6] в некотором смысле преждевременна.

Одной из самых сильных и самых слабых сторон Graphite является то, что он, в действительности, очень мало «проектировался» в традиционном смысле слова. По большому счету Graphite развивался постепенно и брал препятствия по мере того, как возникали проблемы. Много раз препятствие можно было предвидеть и различные упреждающие решения казались естественными. Однако, было бы лучше избегать решения проблем, с которыми вы еще не столкнулись, даже если кажется, что они вскоре возникнут. Причина в том, что вы можете узнать гораздо больше от внимательного изучения фактических неудач, чем при теоретизировании о превосходной стратегии. Решение проблемы обуславливается как эмпирическими данными, которые у нас есть под рукой, так и нашими собственными знаниями и интуицией. Я узнал, что сомнения в достаточности собственных знаний может заставить вас посмотреть более тщательно на ваши эмпирические данные.

Например, когда я впервые написал `whisper`, я был убеждён, что он должен быть переписан на язык С с тем, чтобы увеличить скорость, и что код на языке Python мог бы служить исключительно в качестве прототипа. Если бы тогда я не был в цейтноте, я бы, скорее всего, полностью исключил применение языка Python. Однако, оказывается, что ввод/вывод становится узким местом гораздо раньше, чем процессор, и что на практике меньшая эффективность языка Python вряд ли вообще имеет значение.

Как я уже говорил, эволюционный подход также является большой слабостью проекта Graphite. Интерфейсы, как оказалось, плохо поддаются постепенной эволюции. Хороший интерфейс для максимальной предсказуемости должен быть согласованным и должен отвечать соглашениям об использовании. По этому показателю, интерфейс URL API в Graphite является, на мой взгляд, в настоящее время неудовлетворительным. С течением времени добавлялись параметры и функции, иногда образовывались небольшие острова согласованности, но в целом общей согласованности не хватает. Единственный способ решить такую проблему является переход к версиям интерфейсов, но здесь тоже есть свои недостатки. Как только разрабатывается новый интерфейс, от старого интерфейса становится трудно избавиться и он сохраняется повсюду как эволюционный багаж, похожий на аппендикс у человека. Это может казаться достаточно безобидным, пока однажды у вашего кода этот аппендикс не воспалится (т.е. Не возникнет ошибка, связанная со старым интерфейсом) и вам придется прибегнуть к операции. Если бы я на ранней стадии должен был изменить в Graphite только одну вещь, то я бы уделил гораздо больше внимания разработке внешних интерфейсов API, обдумывая их заранее, а не собирая их по крупицам.

Другим аспектом Graphite, который вызывает некоторое разочарование, является ограниченная гибкость иерархической модели именования метрик. Хотя она довольно проста и очень удобна в большинстве случаев, с ее помощью становится трудно и даже невозможно делать некоторые сложные запросы. Когда я впервые подумал о создании Graphite, я с самого начал знал а, что для создания графиков хотел иметь интерфейс API на основе URL, который мог бы редактировать человек [7]. Хотя я до сих пор рад, что в Graphite сегодня есть эта возможность, я боюсь, что такое требование заставляет ограничиться в API исключительно простым синтаксисом, который делает сложные выражения громоздкими. Иерархия делает проблему определения «первичного ключа» для метрики достаточно простой, поскольку для узла в дереве первичным ключом, по существу, является путь. Недостаток состоит в том, что все описательные данные (т.е. данные в столбцах) должны встраиваться непосредственно в пути. Возможное решение состоит в поддержке иерархической модели и добавлении отдельной базы метаданных, которая с помощью специального синтаксиса предоставит более усовершенствованные способы выборки метрик.

7.11. Переход в статус открытого кода

Оглядываясь на эволюцию Graphite, я все еще удивляюсь тому, как далеко он зашел как проект, и как далеко он завел меня как программиста. Он начался, как любимое занятие, в котором был всего лишь несколько сотен строк кода. Движок отрисовки стартовал в качестве эксперимента просто для того, чтобы я мог увидеть, смогу ли написать его. `whisper` был написан в течение выходных от отчаяния для решения непреодолимой проблемы к критической дате запуска. `carbon` переписывался столько раз, что сложно вспомнить. Когда в 2008 году мне только что разрешили выпустить Graphite под лицензией открытого исходного кода, я не ожидал, что будет отклика. Через несколько месяцев Graphite был упомянут в статье в CNET, что было замечено в Slashdot и проекта вдруг стал популярным и его популярность продолжается до сих пор. Сегодня существуют десятки крупных и средних компаний, использующих Graphite. Сообщество достаточно активно и продолжает расти. Проекту далеко до завершения, есть много классной экспериментальной работы, которая выполняется, и это поддерживает интерес к работе и открывает большие перспективы.

Примечания

1. <http://launchpad.net/graphite>
2. Есть еще один порт, через который можно пересыпать сериализированные объекты, что более эффективно, чем пересылка в простом текстовом формате. Это необходимо только для очень высоких объемов трафика.
3. <http://memcached.org>
4. Твердотельные накопители в сравнении обычными жесткими дисками обычно имеют очень большую скорость позиционирования.
5. Файлы RRD в действительности являются узлами-ветками, поскольку в них могут указываться несколько источников данных; источники данных RRD являются узлами-листьями.
6. Кнут, в частности, имел в виду низкоуровневую оптимизацию кода, а не макрооптимизацию, такую как улучшение проекта.
7. Это требует, чтобы сами графики были открыты. Любой может просто посмотреть на URL графика, чтобы понять или изменить его.

8. Распределенная файловая система Hadoop

Глава 8 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Распределенная файловая система Hadoop (The Hadoop Distributed File System - HDFS) спроектирована с учетом возможности надежного хранения чрезвычайно больших объемов данных и их передачи на высокой скорости пользовательским приложениям. В составе кластера большого размера пользовательские приложения выполняются на тысячах серверов, работающих с подключенными напрямую устройствами хранения данных. Распределяя хранилище данных и вычислительные ресурсы между множеством серверов, можно осуществлять наращивание ресурсов при необходимости, делая систему любого размера экономичной. Мы опишем архитектуру распределенной файловой системы HDFS и расскажем о нашем опыте использования данной файловой системы для управления 40 петабайтами данных компании Yahoo!

8.1. Введение

Hadoop¹ является распределенной файловой системой, также включающей в свой состав фреймворк для проведения анализа и преобразований очень больших объемов данных с использованием парадигмы MapReduce [[DG04](#)]. Хотя интерфейс HDFS и проектировался по аналогии с интерфейсами файловых систем из состава Unix, разработчики пожертвовали точностью следования стандартам в угоду повышению производительности используемых приложений.

Важной характеристикой файловой системы Hadoop является распределение данных и вычислительных ресурсов между многими (тысячами) узлов, а также выполнение предусмотренных при-

ложениями вычислений параллельно с доставкой необходимых данных. Кластер Hadoop позволяет масштабировать вычислительные ресурсы, емкость устройств хранения данных и пропускную способность каналов для осуществления операций ввода/вывода путем простого добавления приобретаемых серверов. Кластеры Hadoop в компании Yahoo! в совокупности состоят из 40000 серверов и хранят 40 петабайт данных приложений, при этом самый большой кластер состоит из 4000 серверов. Около ста других организаций со всего мира заявляют об использовании Hadoop.

Файловая система HDFS хранит метаданные и данные приложений отдельно. Как и другие распределенные файловые системы, такие, как PVFS [[CIRT00](#)], Lustre² и GFS [[GGL03](#)], HDFS хранит метаданные на выделенном сервере, называемом сервером метаданных (NameNode). Данные приложений хранятся на других серверах, называемых серверами данных приложений (DataNode). Все серверы взаимодействуют друг с другом посредством протоколов, основывающихся на протоколе TCP. В отличие от файловых систем Lustre и PVFS, используемые HDFS серверы данных приложений (DataNode) не полагаются с целью повышения надежности хранения данных на такие механизмы их защиты, как RAID. Напротив, аналогично файловой системе GFS, содержимое файла распределяется между множеством серверов данных приложений для его надежного хранения. Гарантируя надежность хранения данных, применяемая стратегия позволяет использовать преимущество умножения пропускной способности канала передачи данных, а также в случае ее использования появляется возможность проводить вычисления параллельно с доставкой необходимых данных.

8.2. Архитектура

8.2.1. Сервер метаданных NameNode

Пространство имен файловой системы HDFS представлено иерархией файлов и директорий. Файлы и директории представлены структурами inode на сервере метаданных (NameNode). Структуры inode содержат такие атрибуты файла, как права доступа, метки времени модификации и последнего доступа, квоты для пространства имен и дискового пространства. Содержимое файла разделяется на большие блоки (обычно размером в 128 мегабайт, но этот размер также может задаваться пользователем для каждого из файлов) и каждый такой блок файла независимо копируется на множество серверов данных приложений. Текущая архитектура предусматривает возможность использования одного сервера метаданных (NameNode) для каждого кластера. Кластер может содержать тысячи серверов данных приложений и обслуживать десятки тысяч клиентов HDFS, так как каждый сервер данных приложений позволяет выполнять множество приложений одновременно.

8.2.2. Образ и журнал

Структуры inode и список блоков, являющиеся метаданными системы имен, называются образом. Сервер метаданных (NameNode) хранит образ пространства имен полностью в оперативной памяти. Постоянная копия образа, хранящаяся в локальной файловой системе сервера метаданных называется контрольной точкой. Данные изменений HDFS, сохраняемые в локальной файловой системе сервера метаданных перед произведением самих изменений, называются журналом. Данные о расположении скопированных блоков файлов не являются постоянной частью файла контрольной точки.

Каждая инициированная клиентом транзакция находит свое отражение в журнале, при этом изменения файла журнала принудительно записываются на диск перед отправкой подтверждения клиенту. Файл контрольной точки никогда не изменяется по инициативе сервера метаданных; новый файл записывается тогда, когда контрольная точка создается в ходе перезагрузки, при запросе от администратора или с участием сервера файлов контрольных точек (CheckpointNode), описанного в следующем разделе. В ходе запуска сервера метаданных происходит инициализация образа пространства имен с использованием файла контрольной точки, после чего в образ вносятся измене-

ния из журнала. Новый файл контрольной точки и пустой журнал записываются в файловую систему сервера метаданных перед тем, как он начинает обслуживать клиентов.

Для повышения длительности хранения данных резервные копии файлов контрольной точки и журнала обычно хранятся на множестве независимых локальных разделов и на удаленных NFS-серверах. Первая мера предосторожности предотвращает потерю данных в случае выхода из строя отдельного раздела, вторая - в случае выхода из строя всего сервера. Если сервер метаданных сталкивается с ошибкой при копировании файла журнала в одну из директорий для его хранения, он автоматически исключает эту директорию из списка директорий. В том случае, если ни одна из директорий для хранения файлов не доступна, сервер метаданных автоматически прекращает свою работу.

Сервер метаданных использует систему с поддержкой программных потоков и обрабатывает запросы от множества клиентов одновременно. Операция сохранения данных транзакции на диск становится узким местом, так как всем потокам приходится ожидать завершения процедуры синхронной записи, инициированной одним из них. На самом деле, для оптимизации этого процесса сервер метаданных создает очередь из множества транзакций. Когда один из потоков сервера метаданных инициирует операцию записи, все ожидающие в очереди транзакции, выполняются вместе. Остающиеся потоки должны только проверить, были ли сохранены данные в ходе транзакций и не должны осуществлять принудительную запись.

8.2.3. Сервер данных приложений DataNode

Каждая копия блока файла на сервере данных приложений представлена двумя файлами в локальной файловой системе. Первый файл содержит сами данные, а второй - метаданные блока, включающие в себя контрольные суммы для блока данных и метку времени, относящуюся к моменту генерации блока. Размер файла данных равен используемому размеру блока и файл не занимает дополнительного дискового пространства для округления в сторону повышения размера файла до размера блока таким образом, как это делается в традиционных файловых системах. Следовательно, если размер блока составляет половину установленного размера, он занимает объем локального диска, соответствующий только половине установленного размера блока.

Во время загрузки каждый сервер данных приложений соединяется с сервером метаданных и осуществляет операцию рукопожатия. Целью этой операции рукопожатия является проверка идентификатора пространства имен и версии программного обеспечения на сервере данных приложений. Если хотя бы одно из используемых сервером метаданных значений не совпадает со значением, используемым сервером данных приложений, сервер данных приложений автоматически прекращает свою работу.

Идентификатор пространства имен присваивается файловой системе во время форматирования. Этот идентификатор хранится на всех серверах кластера на постоянной основе. Серверы с различными идентификаторами пространств имен не смогут работать в рамках одного кластера, таким образом защищается целостность файловой системы. Только что инициализированному серверу данных приложений без идентификатора пространства имен разрешено входить в состав кластера и получать идентификатор пространства имен от него.

После рукопожатия сервер данных приложений регистрируется сервером метаданных. Серверы данных приложений постоянно хранят свои уникальные идентификаторы хранилища. Идентификатор хранилища является внутренним идентификатором сервера данных приложений, который позволяет идентифицировать его даже после перезагрузки и присвоения ему другого IP-адреса или номера порта. Идентификатор хранилища присваивается серверу данных приложений при регистрации сервером метаданных в первый раз и никогда не изменяется после этого.

Сервер данных приложений сообщает о принадлежащих ему блоках данным серверу метаданных путем отправки отчетов о блоках. Каждый отчет содержит идентификатор блока, метку времени

генерации блока и размер копии блока на сервере. Первый отчет о блоках отправляется немедленно после регистрации сервером метаданных. Последующие отчеты о блоках отправляются каждый час и предоставляют серверу метаданных актуальную информацию о расположении копий блоков в рамках кластера.

В процессе работы серверы данных приложениями отправляют сообщения о состоянии (heartbeats) серверу метаданных для подтверждения того, что сервер данных приложений работает и копии блоков, хранящиеся на нем, доступны. Стандартным интервалом отправки сообщений о состоянии является трехсекундный интервал. В том случае, если сервер метаданных не получает сообщений состояний от сервера данных приложений в течение десяти минут, он считает, что сервер данных приложения не функционирует и копии блоков данных, хранившиеся на нем, недоступны. После этого сервер метаданных планирует операции создания новых копий хранящихся на отключенном сервере блоков на других серверах данных приложений.

Сообщения о состоянии серверов данных приложений также содержат информацию об общем объеме устройств хранения, процентном отношении использованного дискового пространства и количестве передач данных, осуществляемых в текущий момент. Эти статистические данные используются сервером метаданных для принятия решений о резервировании блоков и балансировке нагрузки.

Сервер метаданных не отправляет запросы напрямую серверам данных приложений. Он использует ответы на сообщения о состоянии для отправки инструкций серверам данных приложений. Инструкции включают в себя команды для копирования блоков на другие серверы, удаления локальных копий блоков, повторной регистрации, немедленной отправки отчетов о блоках и завершения работы сервера.

Эти команды особенно важны для поддержания целостности системы, следовательно критически важно поддерживать постоянную отправку сообщений о состоянии даже в крупных кластерах. Сервер метаданных может обрабатывать тысячи сообщений о состоянии в секунду без нарушения процесса выполнения других операций.

8.2.4. Клиент HDFS

Пользовательские приложения получают доступ к файловой системе с помощью клиента HDFS, библиотеки, экспортирующей интерфейс файловой системы HDFS.

Как и большинство традиционных файловых систем, HDFS поддерживает операции чтения, записи и удаления файлов, а также операции создания и удаления директорий. Пользователь описывает файлы и директории с помощью путей из пространства имен. Пользовательскому приложению не нужно беспокоиться о том, что метаданные и данные файлов из файловой системы хранятся на разных серверах или о том, что блоки имеют множество копий.

Когда приложение читает файл, клиент HDFS в первую очередь запрашивает у сервера метаданных список серверов данных приложений, хранящих копии блоков данных необходимого файла. Список сортируется на основании дистанции до клиента в рамках топологии сети. Клиент соединяется напрямую с сервером данных приложений и запрашивает передачу необходимого блока. Когда клиент осуществляет запись, он в первую очередь требует от сервера метаданных выбора серверов данных приложений для хранения копий первого блока файла. Клиент организует канал между несколькими серверами и отправляет данные. Когда первый блок передан, клиент запрашивает выбор следующих серверов данных приложений для хранения копий следующего блока. Организуется новый канал и клиент отправляет данные следующего блока. Выбор серверов данных приложений для каждого блока вероятнее всего будет различным. Взаимодействия между клиентом, сервером метаданных и серверами данных приложений отражены на Рисунке 8.1.



Рисунок 8.1: Клиент HDFS создает новый файл

В отличие от традиционных файловых систем, HDFS предоставляет API, позволяющий выявить расположения блоков данных файлов. Это обстоятельство позволяет таким приложениям, как фреймворк MapReduce планировать выполнение задач на тех серверах, где располагаются необходимые данные, тем самым повышая скорость чтения данных. Обычно файлы подвергаются трехкратной репликации. В случае работы с важными файлами или файлами, доступ к которым осуществляется очень часто, повышенная степень репликации повышает устойчивость к отказам и скорость чтения данных.

8.2.5. Сервер файлов контрольных точек CheckpointNode

Сервер метаданных в HDFS в дополнение к его основным задачам обработки клиентских запросов, может работать в одном из двух других режимов, выполняя функции сервера файлов контрольных точек (CheckpointNode) или сервера резервных копий (BackupNode). Режим работы устанавливается в процессе загрузки сервера.

Сервер файлов контрольных точек периодически комбинирует данные из существующего файла контрольной точки с данными из журнала для создания нового файла контрольной точки и пустого журнала. Сервер файлов контрольных точек обычно работает на отдельном от сервера метаданных узле, так как имеет аналогичные требования к оперативной памяти. Он загружает текущие файлы контрольной точки и журнала с сервера метаданных, объединяет их и возвращает новый файл контрольной точки серверу метаданных.

Периодическое создание файлов контрольных точек является одним из методов защиты метаданных файловой системы. Система может начать работу с наиболее поздним файлом контрольной точки в том случае, если все остальные постоянные копии образа пространства имен и журнала недоступны. Создание файла контрольной точки также позволяет серверу метаданных очистить журнал в тот момент, когда загружается новый файл контрольной точки. Кластеры HDFS работают в течение длительных промежутков времени без перезагрузок, при этом файл журнала также постоянно растет в течение этих промежутков времени. Если файл журнала достигнет очень большого объема, повысится вероятность его потери или повреждения. Также, очень большой файл журнала увеличивает период времени, требуемый для перезагрузки сервера метаданных. При работе большого кластера обработка файла журнала с событиями за неделю происходит в течение часа. Хорошей практикой является ежедневное создание файла контрольной точки.

8.2.6. Сервер резервных копий BackupNode

Недавно была представлена возможность использования сервера резервных копий (BackupNode) в HDFS. Как и в случае сервера файлов контрольных точек, сервер резервных копий позволяет периодически создавать файлы контрольных точек, но в дополнение к этому он поддерживает в оперативной памяти актуальный образ пространства имен файловой системы, который может быть синхронизирован с соответствующим образом сервера метаданных.

Сервер резервных копий принимает поток транзакций из пространства имен в форме файлов журнала от активного сервера метаданных, сохраняет свою копию журнала и применяет эти транзакции по отношению к своему образу пространства имен в оперативной памяти. Сервер метаданных рассматривает сервер резервных копий как хранилище файлов журнала, аналогично хранилищу файлов журнала в локальных директориях. В случае неработоспособности сервера метаданных, образ пространства имен в оперативной памяти сервера резервных копий и файл контрольной точки на его диске будут содержать актуальные данные состояния пространства имен файловой системы.

Сервер резервных копий может создать файл контрольной точки без загрузки файлов контрольной точки и журнала с активного сервера метаданных, так как он и так имеет в распоряжении актуальный образ пространства имен файловой системы в оперативной памяти. Это обстоятельство делает процесс создания файла контрольной точки сервером резервных копий более эффективным, так как ему требуется только сохранить образ пространства имен в свои локальные директории.

Сервер резервных копий может рассматриваться как сервер метаданных, работающий в режиме только для чтения. Он хранит все метаданные файловой системы за исключением расположений блоков данных файлов. Он может выполнять все функции обычного сервера метаданных, не связанные с модификацией пространства имен или доступом к информации о расположении блоков данных файлов. Использование сервера резервных копий позволяет использовать сервер метаданных без постоянного хранилища, делегируя ответственность за постоянное хранение состояния пространства имен серверу резервных копий.

8.2.7. Обновления и снимки файловой системы

В процессе обновлений программного обеспечения вероятность повреждения файловой системы в результате ошибок программного обеспечения или обслуживающего персонала возрастает. Целью создания снимков файловой системы HDFS является минимизация потенциальных рисков повреждения данных системы во время ее обновлений.

Механизм снимков позволяет системным администраторам сохранить состояние файловой системы в неизменном виде, поэтому в том случае, если обновление системы приведет к потере данных или их повреждению, будет возможность отката изменений, вызванных обновлением системы, и возвращения пространства имен и состояния данных файловой системы HDFS к тому виду, в каком они были во время создания снимка.

Снимок (который может существовать только в единственном экземпляре) создается по усмотрению администратора кластера в любой момент после запуска системы. При запросе создания снимка сервер метаданных сначала считывает данные из файлов контрольной точки и журнала, объединяя их в оперативной памяти. После этого он записывает новые файлы контрольной точки и пустого журнала в других директориях таким образом, чтобы старые файлы контрольной точки и журнала не подверглись изменениям.

Во время обмена данными сервер метаданных отправляет серверам данных приложений команду создания локальных снимков. Локальный снимок на сервере данных приложений не может быть создан с помощью репликации директорий с файлами данных, так как эта операция потребовала бы удвоения объема устройств хранения данных каждого сервера данных приложений кластера. Вместо этого каждый сервер данных приложений создает копию директории для хранения данных, содержащую жесткие ссылки на существующие блоки данных. Когда сервер данных приложений удаляет блок, он удаляет только жесткую ссылку, а при модификациях блока в ходе поступления данных используется техника копирования при записи. Таким образом, копии старых блоков остаются нетронутыми в старых директориях.

Администратор кластера может осуществить откат файловой системы HDFS к состоянию в момент создания снимка при перезагрузке системы. Сервер метаданных восстановит файл контроль-

ной точки, сохраненный во время создания снимка. Серверы данных приложений восстанавливают ранее переименованные директории и инициируют фоновый процесс удаления копий блоков, созданных после создания снимка. После использования функции отката не остается возможности для возврата в предыдущее состояние. Администратор кластера может освободить место, занятое данными снимка, использовав команду удаления снимка; для снимков, созданных во время обновления программного обеспечения эта операция завершается после обновления системы.

Эволюция системы может повлечь изменения в формате файлов контрольной точки и журнала сервера метаданных или представления копий файлов блоков на серверах данных приложений. Версия представления данных идентифицирует метод представления данных и постоянно хранится в директориях для файлов сервера метаданных и серверов данных приложений. В ходе загрузки каждый сервер сравнивает версию представления данных, используемую программным обеспечением, с версией представления данных, хранящихся в директориях, и автоматически преобразует данные из представлений устаревших форматов в новые. Преобразование требует обязательного создания снимка после перезагрузки сервера с использованием программного обеспечения, поддерживающего новую версию представления данных.

8.3. Операции ввода/вывода и управление копиями блоков

Конечно же, задачей файловой системы является хранение данных в файлах. Для понимания того, как эта задача реализуется файловой системой HDFS, нам следует рассмотреть процесс чтения и записи, а также процесс управления блоками данных.

8.3.1. Чтение и запись файлов

Приложение добавляет данные в файловую систему HDFS, создавая новый файл и записывая данные в него. После того, как файл закрывается, записанные байты не могут быть изменены или удалены, за исключением случаев, когда новые данные могут быть добавлены в файл после его повторного открытия для дополнения. Файловая система HDFS реализует модель, в рамках которой может функционировать один записывающий и множество читающих данные процессов.

Клиент HDFS, открывающий файл для записи, получает файл в свое полное распоряжение; ни один из других клиентов не сможет осуществить запись в этот файл. Записывающий данные клиент периодически подтверждает актуальность процесса модификации файла, отправляя сообщения о состоянии серверу метаданных. Когда файл закрывается, блокировка записи другими процессами снимается. Длительность блокировки записи другими процессами ограничивается мягким и жестким лимитами времени. До того, как мягкий лимит времени истекает, для записывающего данные процесса гарантируется эксклюзивный доступ к файлу. Если мягкий лимит времени истечет и клиент не закроет файл и не подтвердит блокировку с помощью сообщения о состоянии, блокировка может быть установлена для другого клиента. Если по истечении жесткого лимита времени (длительностью в один час) клиент не подтвердит блокировку, файловая система HDFS посчитает, что клиент завершил свою работу и автоматически закроет файл вместо клиента, устранив блокировку. Блокировка файла записывающим данным процессом не запрещает другим клиентам читать файл; файл может читаться параллельно множеством процессов.

В файловой системе HDFS файл состоит из блоков. Когда требуется новый блок, сервер метаданных резервирует блок с уникальным идентификатором и создает список серверов данных приложений для хранения копий блока. Серверы данных приложений формируют канал для передачи данных в порядке, который минимизирует общую дистанцию от клиента до наиболее удаленного сервера данных приложений. Байты передаются в канал в виде последовательности пакетов. Байты, которые приложение записывает в файл, в первую очередь подвергаются буферизации на стороне клиента. После заполнения буфера пакета (обычно размером в 64 КБ) данные передаются в канал. Следующий пакет может быть передан в канал перед приемом подтверждения доставки

предыдущих пакетов. Количество не доставленных пакетов ограничивается размером окна для пакетов на стороне клиента.

После того, как все данные записаны в файл HDFS, файловая система не предоставляет никаких гарантий, что данные будут доступны новым процессам, открывающим файл для чтения, до момента его закрытия. Если пользовательскому приложению требуется гарантия того, что данные будут доступны для чтения, оно может явно выполнить операцию `hflush`. После этого текущий пакет немедленно отправляется в канал для передачи данных и операция `hflush` будет ожидать того момента, когда все серверы данных приложений, использующие канал, подтвердят успешный прием пакета. После этого все данные, записанные перед выполнением операции `hflush`, будут гарантированно доступны для чтения.

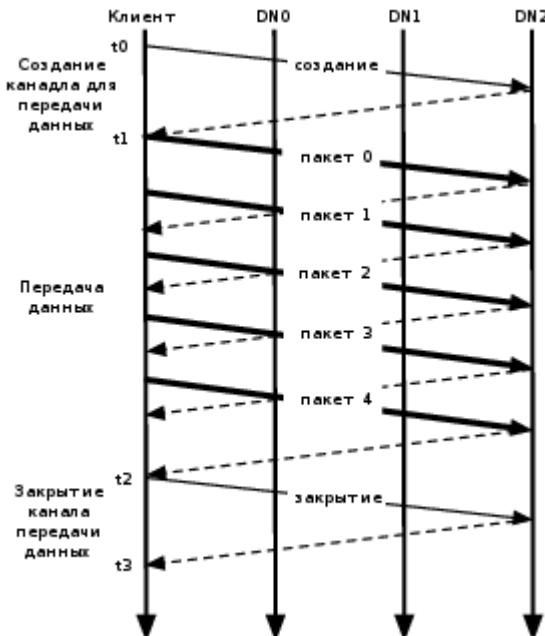


Рисунок 8.2: Состояние канала передачи данных во время записи блока

В случае отсутствия ошибок создание блока происходит в течение трех стадий, так, как показано на Рисунке 8.2, иллюстрирующем канал передачи данных из трех серверов данных приложений (DN) и блока из пяти пакетов. На рисунке с помощью жирных линий изображены пакеты данных, с помощью штриховых - сообщения для подтверждения передачи, а с помощью обычных линий - управляющие сообщения для создания и закрытия канала. Вертикальные линии отображают активность клиента и трех серверов данных приложений, причем время отсчитывается сверху вниз. В течение промежутка времени от t_0 до t_1 длится стадия создания канала для передачи данных. В интервале времени от t_1 до t_2 - стадия передачи данных, где t_1 является временем отправки первого пакета, а t_2 - время приема подтверждения доставки последнего пакета. В данном случае при передаче пакета 2 используется операция `hflush`. Указатель использования операции `hflush` передается с данными пакета и не используется в составе отдельной операции. Последний интервал времени от t_2 до t_3 соответствует стадии закрытия канала передачи данных для этого блока.

В кластере из тысяч серверов выходы из строя серверов (чаще всего из-за выхода из строя устройства хранения данных) случаются ежедневно. Копии блоков, хранящиеся на сервере данных приложений, могут быть повреждены из-за неисправностей оперативной памяти, диска или сети. Файловая система HDFS генерирует и хранит контрольные суммы для каждого из блоков данных файла HDFS. Контрольные суммы сверяются клиентом HDFS во время чтения файла для установления факта любого повреждения, вызванного клиентом и серверами данных приложений, либо сетью. Когда клиент создает файл HDFS, он вычисляет последовательность контрольных сумм для каждого блока и отправляет их серверу данных приложений вместе с данными. Сервер данных приложений сохраняет контрольные суммы в отдельном от блока данных файле метаданных. Когда HDFS читает файл, контрольные суммы каждого блока доставляются клиенту. Клиент вычис-

ляет контрольную сумму принятых данных и проверяет, совпадают ли полученные контрольные суммы с рассчитанными. Если контрольные суммы не совпадают, клиент сообщает серверу метаданных о поврежденной копии блока, после чего принимает другую копию блока от другого сервера данных приложений.

Когда клиент открывает файл для чтения, он получает список блоков и данные о размещении каждой из копий блоков от сервера метаданных. Данные о размещении каждого блока расположены в зависимости от их дистанции от сервера, на котором осуществляется чтение. При чтении содержимого блока клиент в первую очередь пробует принять наиболее близко расположенную копию. Если попытка чтения не удаётся, клиент пытается прочитать следующую копию из последовательности. Чтение может завершиться неудачей в случае, если целевой сервер данных приложений не доступен, сервер больше не хранит копию блока или копия блока считается поврежденной после сравнения контрольных сумм.

Файловая система позволяет клиенту читать открытый для записи файл. При чтении открытого для записи файла длина последнего все еще записываемого блока неизвестна серверу метаданных. В этом случае клиент получает данные одной из копий для получения последнего значения размежа перед началом чтения содержимого.

Архитектура системы ввода/вывода файловой системы HDFS особым образом оптимизирована для систем последовательной обработки, таких, как MapReduce, требующих высокой скорости передачи данных при последующих операциях чтения и записи. Продолжающиеся оптимизации должны улучшить время чтения/записи для приложений, требующих передачи данных в реальном времени или случайного доступа к данным.

8.3.2. Размещение блоков

При создании кластера большого размера использование плоской топологии для соединения серверов может оказаться непрактичным. Обычной практикой является установка серверов в множестве стоек. Серверы в стойке совместно используют свитч, а свитчи стоек соединены с помощью одного или нескольких центральных свитчей. Взаимодействие двух серверов из различных стоек происходит в результате преодоления данными множества свитчей. В большинстве случаев трафик между серверами из одной стойки превышает трафик между серверами из разных стоек. На Рисунке 8.3 изображен кластер, состоящий из двух стоек, каждая из которых содержит по три сервера.

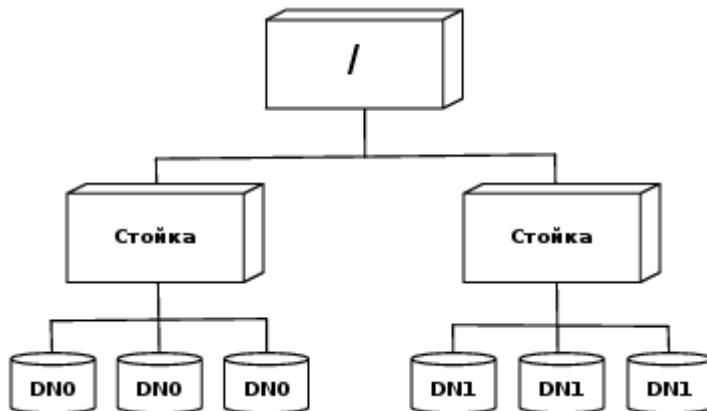


Рисунок 8.3. Топология кластера

Файловая система HDFS оценивает интенсивность трафика между двумя серверами на основании их удаления друг от друга. Удаление сервера от своего родительского сервера принимается за единицу отсчета. Расстояние между двумя серверами может быть рассчитано как сумма расстояний до их ближайших родительских серверов. Более короткое расстояние между двумя серверами подразумевает возможность повышения интенсивности трафика.

Файловая система позволяет администратору использовать сценарий, который будет возвращать информацию о принадлежности сервера к стойке на основе адреса этого сервера. Сервер метаданных является центральной точкой определения расположения стойки для каждого сервера данных приложений. Когда происходит регистрация сервера данных приложений сервером метаданных, данный сервер выполняет сценарий для определения принадлежности сервера данных приложений к конкретной стойке. Если такой сценарий не используется, сервер метаданных считает, что все серверы данных приложений расположены в одной стандартной стойке.

Размещение копий блоков является критичным параметром для надежности хранения данных и скорости чтения и записи файловой системы HDFS. Удачная политика размещения копий блоков должна улучшить надежность хранения данных, доступность данных и оптимизировать использование сети. На данный момент HDFS предоставляет интерфейс настройки политики размещения блоков, поэтому пользователи и исследователи могут экспериментировать и тестировать альтернативные политики, оптимальные для их приложений.

Стандартная политика размещения блоков HDFS является компромиссным решением, обеспечивающим баланс между минимизацией затрат ресурсов для записи данных и максимизацией доступности данных, надежности их хранения и общей скоростью чтения. При создании нового блока HDFS размещает первый блок на том сервере, на котором выполняется осуществляющий запись процесс. Вторая и третья копии располагаются на двух различных серверах в другой стойке. Остальные копии размещаются на случайных серверах с учетом условий, согласно которым на каждом сервере может располагаться не более одной копии блока и не более двух копий могут располагаться на серверах одной стойки, если это возможно. Выбор для размещения второй и третьей копии блока различных серверов обуславливает лучшее распространение копий блоков одного файла в пределах кластера. Если две первые копии блоков файла располагаются на серверах одной и той же стойки, то для любого файла две трети копий блока будут также располагаться на серверах одной стойки.

После того, как выбраны все целевые серверы, между ними организуется канал передачи данных с учетом их удаления от сервера с первой копией блока. Данные передаются всем серверам в заданной последовательности. Для чтения данных сервер метаданных в первую очередь проверяет, находится ли клиентский узел в рамках кластера. Если это условие выполняется, информация о расположении блоков, отсортированная по близости к узлу, на котором осуществляется чтение данных, передается клиенту. Чтение блока с серверов данных приложений осуществляется в этом же порядке.

Эта политика позволяет снизить интенсивность трафика между стойками и между узлами и в общем случае повысить скорость записи данных. Так как вероятность выхода из строя стойки значительно ниже вероятности выхода из строя сервера, эта политика не влияет на гарантии сохранности и доступности данных. В обычном случае использования трех копий данная политика позволяет снизить суммарную интенсивность трафика при чтении данных, так как блок располагается только на серверах из двух различных стоек вместо трех.

8.3.3. Управление репликаций

Сервер метаданных пытается убедиться в том, что каждый блок всегда имеет заданное количество копий. Он устанавливает факт недостатка копий или наличия излишних копий в момент доставки отчета о блоках от серверов данных приложений. В случаях, когда блок имеет лишние копии, сервер метаданных выбирает копию для удаления. Он предпочтет не сокращать количество стоек, на которых хранятся копии, а также предпочтет удалить копию с того сервера данных приложений, на котором наименьшее количество доступного дискового пространства. Целью данной политики является балансировка использования устройств хранения данных серверов данных приложений без снижения доступности блоков.

Когда количество копий блока становится недостаточным, он попадает в очередь приоритетной репликации. Блок только с одной копией имеет наивысший приоритет, а блок с несколькими копиями, составляющими более двух третьих необходимого объема репликации - низший. Программный поток, работающий в фоновом режиме, периодически сканирует начало очереди репликации для принятия решения о том, где размещать новые копии. Процесс репликации блоков использует простую политику размещения новых копий блоков. Если доступна одна копия блока, HDFS размещает следующую копию на сервере из другой стойки. В случае, когда блок имеет две доступных копии, если две существующие копии находятся на серверах одной стойки, третья копия создается на сервере из другой стойки; в противном случае третья копия размещается на другом сервере той же стойки, используемой существующей копией. В данном случае целью политики является снижение затрат на создание новых копий.

Сервер метаданных также осуществляет контроль с целью недопущения размещения всех копий блока на серверах одной стойки. Если сервер метаданных устанавливает факт размещения всех копий блока на серверах одной стойки, он считает, что у блока недостаточно копий и создает копию блока на сервере из другой стойки, используя такую же политику размещения блоков, описанную выше. После того, как сервер метаданных получает уведомление о завершении создания копии, блок считается подвергнутым излишней репликации. Впоследствии сервер метаданных решает удалить старую копию, так как политика управления копиями предусматривает действия, не направленные на снижение количества серверов с копиями блоков в различных стойках.

8.3.4. Балансировщик

Стратегия размещения блоков файловой системы HDFS не учитывает использование дискового пространства серверами данных приложений. Она используется для запрета размещения новых, наиболее вероятно используемых, данных на небольшом множестве серверов данных приложений с большим количеством свободного дискового пространства. Следовательно, данные могут не всегда равномерно размещаться на серверах данных приложений. Дисбаланс также возникает при добавлении новых серверов в кластер.

Балансировщик является инструментом, позволяющим достигать равномерного использования дискового пространства серверами кластера HDFS. В качестве исходных данных используется пороговое значение, являющееся дробным числом из диапазона от 0 до 1. Кластер считается сбалансированным в том случае, когда степень использования диска каждого сервера данных приложений³ отличается от степени использования диска всеми серверами кластера⁴ не больше, чем на пороговое значение.

Данный инструмент функционирует в виде приложения, которое может быть запущено администратором кластера. Оно последовательно перемещает копии блоков с серверов данных приложений, на которых чрезмерно используется дисковое пространство, на сервера данных приложений, дисковое пространство которых не используется в достаточной мере. Единственным ключевым требованием к балансировщику является поддержание доступности данных. При выборе копии для перемещения и определении направления перемещения, балансировщик гарантирует, что данное решение не снизит количество копий блока и не уменьшит количество стоек, используемых серверами для хранения копий.

Балансировщик оптимизирует процесс перемещения данных, минимизируя копирование данных между серверами из различных стоек. Если балансировщик решает, что копия А должна быть перемещена на сервер в другой стойке и сервер из этой стойки уже содержит копию В этого же блока, будет использована копия В вместо копии А.

Параметр настройки ограничивает интенсивность трафика, генерируемого в ходе операций ребалансировки. Чем выше интенсивность трафика, тем быстрее кластер достигнет сбалансированного состояния, но это происходит в ущерб скорости работы приложений.

8.3.5. Сканер блоков

На каждом сервере данных приложений работает сканер блоков, который периодически сканирует хранящиеся на сервере копии блоков и проверяет соответствие данных блоков их сохраненным контрольным суммам. В течение каждого периода сканирования сканер блоков устанавливает допустимую интенсивность трафика для завершения процесса проверки блоков в течение заданного периода времени. Если клиент читает блок полностью и проверка его данных на соответствие контрольной сумме завершается успешно, он информирует об этом сервер данных приложений. После этого сервер данных приложений считает, что проверка блока прошла успешно.

Время проверки каждого блока сохраняется в журнале событий в понятном человеку формате. В любой момент времени в директории верхнего уровня сервера данных приложений хранится до двух файлов, являющихся используемым в данный момент и ранее журналами событий. Новые записи о времени проверки добавляются в используемый файл журнала событий. Соответственно, каждый сервер данных приложений хранит в памяти список копий блоков для сканирования, отсортированный по времени их последней проверки.

Всякий раз, когда читающий данные клиент или сканер блоков обнаруживает поврежденный блок, он оповещает об этом сервер метаданных. Сервер метаданных отмечает копию как поврежденную, но не планирует удаление этой копии незамедлительно. Вместо этого он начинает репликацию неповрежденной копии блока. Только тогда, когда количество неповрежденных копий блока достигает необходимого количества для данного блока, планируется удаление поврежденной копии. Целью этой политики является сохранение данных в течение такого долгого периода, как это возможно. Таким образом, даже если все копии блока будут повреждены, используемая политика позволит пользователю получить поврежденные данные копий.

8.3.6. Прекращение эксплуатации серверов данных приложений

Администратор кластера формирует список серверов, которые должны быть выведены из эксплуатации. Как только сервер помечен как выводимый из эксплуатации, он не будет использоваться для размещения новых копий блоков, но будет продолжать обслуживать запросы на чтение блоков. Сервер метаданных начнет планирование репликации блоков выводимого из эксплуатации сервера на другие серверы данных приложений. Как только сервер метаданных установит, что все блоки выводимого из эксплуатации сервера скопированы, сервер будет выведен из эксплуатации. После этого он может быть безопасно удален из кластера, что не приведет к риску снижения доступности данных.

8.3.7. Копирование данных между кластерами

При работе с большими наборами данных, перспективы копирования данных из кластера и в кластер HDFS приводят в уныние. Файловая система HDFS предоставляет инструмент под названием DistCp для копирования больших объемов данных внутри и вне кластера в параллельном режиме. Это задача системы MapReduce; каждый из процессов данной системы копирует часть исходных данных в целевую файловую систему. Фреймворк MapReduce автоматически производит планирование выполнения параллельных задач, обработку ошибок и восстановление работоспособности.

8.4. Практическое использование файловой системы в компании Yahoo!

Кластеры HDFS большого размера в компании Yahoo! включают в свой состав около 4000 серверов. Типичный сервер кластера имеет два четырехядерных процессора Xeon, работающих с тактовой частотой 2.5 ГГц, 4-12 подключенных напрямую жестких диска SATA (каждый объемом в два терабайта), 24 ГБ оперативной памяти и соединение Ethernet со скоростью 1 Гбит/с. Семьдесят

процентов объема дискового пространства выделено для файловой системы HDFS. Остальное пространство зарезервировано для операционной системы (Red Hat Linux), файлов журнала, а также для записи выходных данных системы MapReduce (промежуточные данные системы MapReduce не хранятся в файловой системе HDFS).

Сорок серверов из одной стойки совместно используют IP-свитч. Свитчи стоек подключены к каждому из восьми центральных свитчей. Центральные свитчи позволяют установить соединение между стойками и ресурсами за границами кластера. В каждом кластере сервер метаданных и сервер резервных копий снабжаются объемом оперативной памяти до 64 ГБ; приложения никогда не выполняются на этих серверах. В общем, кластер из 4000 серверов располагает 11 ПБ (петабайт равен 1000 терабайтам) доступного дискового пространства для хранения трехкратно скопированных блоков, при этом 3.7 ПБ дискового пространства доступно для пользовательских приложений. В течение многих лет эксплуатации файловой системы HDFS, серверы из состава кластера улучшили свои характеристики благодаря использованию новых технологий. Новые серверы кластера всегда имели более производительные процессоры, большие объемы дискового пространства и оперативной памяти. Более медленные серверы с меньшими объемами дискового пространства выводились из эксплуатации или перемещались в кластеры, зарезервированные для использования в процессе разработки и тестирования Hadoop.

На примере кластера большого размера (из 4000 серверов) можно рассмотреть процесс обслуживания 65 миллионов файлов и 80 миллионов блоков. Так как каждый блок обычно подвергается трехкратной репликации, каждый сервер данных приложений хранит 60000 копий блоков. Каждый день пользовательские приложения создают по два миллиона новых файлов на каждом кластере. Кластер из 40000 серверов компании Yahoo! позволяет использовать сетевое хранилище объемом 40 ПБ.

Переход проекта в разряд ключевых компонентов набора технологий компании Yahoo! подразумевает появление ряда технических проблем, возникающих из-за отличия исследовательского проекта от проекта, отвечающего за сохранность множества петабайт корпоративных данных. Прежде всего эти проблемы связаны с устойчивостью работы системы и надежностью хранения данных. Но также важны и экономическая эффективность, возможность совместного использования ресурсов членами сообщества пользователей и простота обслуживания администраторами системы.

8.4.1. Долговечность хранения данных

Трехкратная репликация данных является надежной мерой, направленной против потери данных в результате непредсказуемых отказов серверов. Едва ли компания Yahoo! когда-нибудь пострадала от потери блока в таких обстоятельствах; для кластера большого размера значение вероятности потери блока в течение одного года меньше 0.005. Ключевым для понимания является тот факт, что около 0.8 процентов серверов выходят из строя в течение каждого месяца. (Даже если работоспособность сервера в конечном счете будет восстановлена, попытка восстановления хранящихся на нем данных обычно не предпринимается.) Таким образом, описанный в примере выше кластер больших размеров теряет каждый день один или два сервера. Этот же кластер повторно создаст копии 60000 блоков, хранящихся на отказавшем сервере, примерно за две минуты: повторная репликация проходит быстро, так как она выполняется в параллельном режиме и масштабируется в зависимости от размера кластера. Вероятность отказа нескольких серверов с копиями одного блока в течение двух минут, приводящего к утрате данных блока, достаточно низка.

Непредсказуемый отказ множества серверов является другой угрозой. В данном случае наиболее часто наблюдаемым является отказ свитча стойки или центрального свитча. Файловая система HDFS допускает потерю свитча стойки (каждый блок имеет копию на сервере из другой стойки). Некоторые неисправности центрального свитча могут действительно привести к отсоединению части стоек от кластера и в таком случае часть блоков может оказаться недоступной. В любом случае, восстановление работы свитча позволяет восстановить недоступные для кластера копии

блоков. Другим непредсказуемым типом отказа является случайное или преднамеренное отключение электроснабжения кластера. Если отключение электроснабжения затрагивает стойки, скорее всего некоторые блоки станут недоступны. Но восстановление электроснабжения не избавит от проблем, так как половина процента серверов не переживет перезагрузку в результате отключения электроснабжения. По статистике, которая подтверждается на практике, кластер большого размера будет терять часть блоков в результате перезагрузки из-за отключения электроснабжения.

В дополнение к отказу серверов, хранимые данные также могут быть повреждены или потеряны. Сканер блоков сканирует все блоки кластера большого размера раз в две недели и находит около 20 поврежденных копий блоков в ходе сканирования. Поврежденные блоки заменяются в момент их обнаружения.

8.4.2. Возможности совместного использования ресурсов HDFS

По мере роста масштабов эксплуатации файловой системы HDFS, в нее были добавлены возможности для совместного использования ресурсов большим количеством отдельных пользователей. Первой такой возможностью был фреймворк прав доступа, спроектированный в соответствии со схемой прав доступа Unix для файлов и директорий. В этом фреймворке файлы и директории имели отдельные права доступа для владельца, других членов группы, ассоциированной с данным файлом или директорией, а также для других пользователей. Принципиальным отличием между правами доступа Unix (POSIX) и HDFS является отсутствие у обычных файлов в HDFS разрешений на исполнение и битов "sticky".

В ранних версиях HDFS применялся слабый механизм идентификации: ваше имя сообщалось серверу узлом, с помощью которого вы подключались. При доступе к HDFS клиентское приложение должно предоставить системе имен идентификационные данные, полученные из доверенного источника. Использование других служб управления идентификационными данными также возможно; начальная реализация использует Kerberos. Пользовательское приложение может использовать тот же фреймворк для подтверждения того, что система имен также является подлинной. Также система имен может запросить идентификационные данные у любого сервера данных приложений из состава кластера.

Общий объем доступного дискового пространства задается количеством серверов данных приложений и дисковым пространством каждого из этих серверов. Опыт эксплуатации ранних версий HDFS показал необходимость введения политики ограничения использования ресурсов группами пользователей. Данная политика должна быть введена не только для справедливого распределения ресурсов, но и для защиты от приложений, записывающих данные на множество узлов и исчерпывающих системные ресурсы, что тоже очень важно. В случае с HDFS метаданные всегда хранятся в оперативной памяти, поэтому размер пространства имен (зависящий от количества файлов и директорий) является также конечным ресурсом. Для управления ресурсами хранилища и пространства имен каждой директории должна соответствовать квота, устанавливающая общее дисковое пространство, которое может быть занято файлами в дереве поддиректорий пространства имен, начинающегося с этой директории. Отдельная квота также может быть установлена для общего количества файлов и директорий в этом дереве поддиректорий.

Архитектура файловой системы HDFS предусматривает тот факт, что большинство приложений будут передавать большие объемы данных, а программный фреймворк MapReduce чаще всего генерирует множество небольших результирующих файлов (по одному для каждой задачи), еще больше исчерпывая ресурсы пространства имен. Для удобства дерево директорий может быть упаковано в единственный файл архива Hadoop (Hadoop Archive file). Файл HAR аналогичен привычным файлам архивов tar, JAR или Zip, но операции файловой системы могут применяться к индивидуальным файлам этого архива, а также файл HAR может прозрачно использоваться в качестве исходного файла для выполнения задачи MapReduce.

8.4.3. Масштабирование и объединение файловой системы

Масштабирование сервера метаданных было ключевой задачей при разработке [Shv10]. Так как сервер метаданных хранит пространство имен и данные о расположении блоков в оперативной памяти, объем доступной памяти сервера метаданных ограничивает количество файлов, а также количество адресуемых блоков. Также данное обстоятельство ограничивает дисковое пространство, которое может обслуживаться сервером метаданных. Пользователям предлагается создавать файлы большего размера, но это предложение не выполняется, так как оно требует изменений в поведении приложений. Более того, мы встречаем новые классы приложений для HDFS, которым требуется хранить большое количество файлов малого размера. Для управления использованием диска были добавлены квоты, а также инструмент архивирования, но они не предоставляют фундаментального решения проблемы масштабирования.

Новая функция позволяет использовать несколько независимых пространств имен (и серверов метаданных) для разделения между ними физического дискового пространства кластера. Пространства имен используют блоки, сгруппированные в пул блоков (Block Pool). Пулы блоков аналогичны логическим единицам (LUN) системы хранилищ SAN, а пространство имен вместе с пулом блоков аналогично разделу файловой системы.

Этот подход имеет массу достоинств помимо расширения возможностей масштабирования: появляется возможность изоляции пространства имен различных приложений для улучшения доступности всего кластера. Абстракция в форме пула блоков позволяет другим службам использовать хранилище блоков, возможно, с другой структурой пространства имен. Мы планируем исследовать другие подходы для улучшения масштабирования, такие, как хранение только части данных пространства имен в оперативной памяти и использование действительно распределенной реализации сервера метаданных.

Приложения предпочитают использовать единственное пространство имен. Пространства имен могут монтироваться для работы в такой унифицированной форме. Таблица монтирования на стороне клиента является эффективным методом реализации этой идеи в сравнении с таблицей на стороне сервера: она позволяет избежать использования системы удаленных вызовов процедур (RPC) для взаимодействия с таблицей на стороне сервера, а также не является восприимчивой к отказам. Простейшим подходом является создание разделяемого пространства имен для всего кластера; этот подход может быть реализован путем добавления на стороне клиента аналогичной монтируемой таблицы для каждого клиента кластера. Таблицы монтирования на стороне клиента также позволяют приложениям создавать частные пространства имен. Эти пространства аналогичны пространствам имен процессов, используемых для работы с технологиями удаленного исполнения в распределенных системах [PPT+93, Rad94, RP93].

8.5. Выученные уроки

Небольшая команда смогла разработать файловую систему Hadoop и сделать ее достаточно стабильной и надежной для промышленной эксплуатации. Успех в большей степени был достигнут благодаря ее чрезвычайно простой архитектуре: копируемым блокам, периодическим отчетам о состоянии блоков и центральному серверу метаданных. Отход от точного следования семантикам POSIX также помог. Хотя решение о хранении всех метаданных в оперативной памяти и ограничило возможности масштабирования пространства имен, оно позволило достаточно просто реализовать сервер метаданных: удалось избежать использования сложных механизмов блокировок, применяемых в стандартных файловых системах. Другой причиной успеха файловой системы Hadoop было ее незамедлительное промышленное использование в компании Yahoo!, так как это обстоятельство позволило быстро и последовательно вносить улучшения. Файловая система очень надежна и нарушения работы сервера метаданных происходят очень редко; на самом деле большая часть периодов неработоспособности связана с обновлениями программного обеспечения. Только недавно в файловой системе были применены (описанные в руководстве) решения, позволяющие повысить отказоустойчивость.

Многие были удивлены выбором языка Java для создания масштабируемой файловой системы. Хотя из-за использования Java и пришлось столкнуться с проблемами масштабирования сервера метаданных ввиду дополнительных затрат памяти на объекты и механизма сборки мусора, Java обуславливает надежность системы; использование этого языка позволяет избежать ошибок, связанных с указателями и управлением памятью.

8.6. Благодарности

Мы хотим поблагодарить компанию Yahoo! за инвестиции в разработку Hadoop и распространение открытого исходного кода; 80% кода HDFS и MapReduce было разработано в компании Yahoo! Мы благодарим всех разработчиков и сотрудников, имеющих отношение к Hadoop, за их весомый вклад в разработку.

Сноски

1. <http://hadoop.apache.org>
2. <http://www.lustre.org>
3. Задается как отношение использованного дискового пространства сервера ко всему доступному дисковому пространству сервера.
4. Задается как отношение использованного дискового пространства кластера ко всему доступному дисковому пространству кластера.

9. Непрерывная интеграция

Глава 9 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Системы непрерывной интеграции (Continuous Integration (CI) systems) предназначены для автоматической и регулярной сборки и тестирования программных продуктов. Хотя их основным преимуществом является возможность устранения длительных периодов времени между сборкой и тестовыми запусками, данные системы также упрощают и автоматизируют выполнение других утомительных задач. Эти задачи включают в себя кроссплатформенное тестирование, регулярное выполнение медленных операций по перемещению больших объемов данных или сложно настраиваемых тестов, проверку достижения необходимой производительности при использовании устаревших платформ, выявление тестов, которые периодически заканчиваются неудачами и регулярное создание пакетов для актуальных версий программных продуктов. Так как автоматизация процессов сборки и тестирования необходима для реализации процесса непрерывной интеграции, данный процесс обычно оказывается первым шагом к реализации фреймворка непрерывного развертывания, в котором обновления программного обеспечения могут производиться в процессе работы систем незамедлительно после тестирования.

Непрерывная интеграция является актуальной темой не только из-за ее значительной роли в гибкой методологии разработки (Agile software methodology). В течение прошедших лет было представлено множество инструментов непрерывной интеграции с открытым исходным кодом, разработанных с использованием и предназначенных для работы с различными языками программирования, реализующих большой диапазон возможностей в контексте набора различных архитектурных моделей. Целью данной главы является описание стандартных наборов возможностей, реализуемых системами непрерывной интеграции, обсуждение доступных архитектурных моделей и установление того, какие возможности можно или нельзя реализовать без лишних сложностей в рамках выбранной архитектуры.

Ниже мы кратко опишем ряд систем, иллюстрирующих различные доступные при проектировании систем непрерывной интеграции архитектурные решения. Первая система, Buildbot использует модель ведущих/ведомых серверов; вторая, Cdash использует модель сервера обработки отчетов; третья, Jenkins, использует гибридную модель; и четвертая, Pony-Build, использует децентрализо-

ванный сервер обработки отчетов, разработанный с использованием языка Python, который мы будем использовать в качестве примера в обсуждении.

9.1. Многообразие систем непрерывной интеграции

В среде архитектур систем непрерывной интеграции прослеживается доминирование двух противоположных моделей: архитектур ведущих/ведомых серверов, в которых центральный сервер осуществляет управление и контролирует работу удаленных серверов для сборки; а также архитектур серверов обработки отчетов, в которых центральный сервер обрабатывает отчеты о сборке, отправленные клиентами. Все системы непрерывной интеграции, о существовании которых нам известно, выбрали некоторую комбинацию возможностей, присущих этим двум архитектурам.

Наш пример централизованной архитектуры, система Buildbot, состоит из двух частей: центрального сервера, также называемого buildmaster и предназначенного для планирования и координации процесса сборки программного обеспечения одним или большим количеством подключенных клиентов; а также клиентов, называемых buildslaves и непосредственно выполняющих сборку. Центральный сервер (buildmaster) является центральной точкой для подключения и хранит информацию о том, какие клиенты должны выполнять команды и в какой последовательности. Клиенты (buildslaves) соединяются с центральным сервером и получают подробные инструкции. Процесс настройки клиента заключается в установке программного обеспечения, идентификации центрального сервера, а также вводе данных для соединения с центральным сервером. Процессы сборки программного обеспечения планируются центральным сервером, а их вывод передается от клиентов центральному серверу и хранится на нем с целью последующего просмотра с помощью веб-интерфейса, а также отправки с помощью других систем отчетов и уведомлений.

На другом крае спектра архитектур находится система CDash, используемая системами виртуализации Visualization Toolkit (VTK)/Insight Toolkit (ITK) от компании Kitware, Inc. На самом деле, система CDash представлена сервером обработки отчетов, спроектированным для хранения и вывода информации, полученной от клиентских компьютеров, выполняющих операции с использованием систем CMake и CTest. При использовании системы CDash клиенты инициируют ряд процессов сборки и тестирования, записывают результаты сборки и тестирования, после чего соединяются с сервером CDash с целью сохранения с помощью него отчета.

Наконец, третья система, Jenkins (известная как Hudson до момента смены названия в 2011 году) может функционировать в двух режимах. В случае использования системы Jenkins, сборка с последующей отправкой отчетов центральному серверу может быть либо инициирована независимо; либо серверы могут исполнять команды центрального сервера Jenkins, который впоследствии будет планировать и управлять ходом процессов сборок.

Централизованная и децентрализованная модели имеют некоторые сходные возможности, и, как мы видим на примере системы Jenkins, эти модели могут сосуществовать в рамках одной реализации. Однако, системы Buildbot и CDash значительно отличаются друг от друга: помимо сходных функций сборки программного обеспечения и создания отчетов о сборке, все остальные аспекты их архитектуры кардинально отличаются. Почему?

Кроме того, в какой степени выбор архитектуры обуславливает простоту или сложность реализации определенных возможностей? Являются ли некоторые возможности следствием использования централизованной модели? И насколько расширяемы возможности существующих реализаций - могут ли они быть без сложностей модифицированы с целью поддержки новых механизмов работы с отчетами, подвергаться масштабированию для работы с множеством пакетов или выполнять операции сборки и тестирования программных продуктов в облачном окружении?

9.1.1. Какие функции выполняются программным обеспечением непрерывной интеграции

Основные функции системы непрерывной интеграции достаточно просты: сборка программного обеспечения, выполнение тестирования и отправка отчета о результатах. Сборка, тестирование и создание отчетов могут осуществляться с помощью сценария, выполнение которого запланировано в форме задачи cron: данный сценарий должен проверить наличие новой копии исходного кода в системе управления версиями (VCS), выполнить сборку исходного кода, после чего выполнить тестирование. Вывод должен быть записан в файл и либо храниться в установленном месте, либо отправляться с помощью электронной почты в случае неполадок. Этот процесс достаточно просто реализуем: в UNIX, например, весь этот процесс для большинства пакетов Python может быть реализован с помощью семистрочного сценария:

```
cd /tmp && \
svn checkout http://some.project.url && \
cd project_directory && \
python setup.py build && \
python setup.py test || \
echo build failed | sendmail notification@project.domain
cd /tmp && rm -fr project_directory
```

На [Рисунке 9.1](#) незаштрихованные прямоугольники представляют отдельные подсистемы и функции в рамках системы. Стрелки указывают направления информационных потоков между различными компонентами. Облако представляет потенциальную возможность удаленного выполнения процессов сборки. Заштрихованные прямоугольники представляют потенциальные объединения подсистем; например, мониторинг процесса сборки включает в себя мониторинг самого процесса сборки, а также мониторинг состояния системы (загрузки центрального процессора, нагрузки вследствие операций ввода/вывода, использования оперативной памяти, и.т.д.)



Рисунок 9.1: Внутреннее устройство системы непрерывной интеграции

Но эта простота обманчива. Реальные системы непрерывной интеграции обычно выполняют намного больше действий. В дополнение к инициированию и приему отчетов о результатах выпол-

нения удаленных процессов сборки, программное обеспечение непрерывной интеграции должно поддерживать любые из следующих дополнительных возможностей:

- *Проверка и обновление исходного кода:* При работе с проектами большого размера создание новой копии исходного кода может быть связано с затратами пропускной способности сети и времени. Обычно системы непрерывной интеграции вместо этого обновляют существующую рабочую копию исходного кода, сталкиваясь при этом только с обработкой изменений со времени предыдущего обновления. В обмен на эту экономию ресурсов, система должна заботиться об обновлении рабочей копии исходного кода и поддерживать ее в актуальном состоянии, что обычно подразумевает хотя бы минимальную интеграцию с системой контроля версий.
- *Абстрактные рецепты сборки:* Рецепт настройки/сборки/тестирования должен быть разработан для использования совместно с интересующим программным обеспечением. Команды низшего уровня обычно различны для различных операционных систем, т.е. для Mac OS X, Windows и UNIX, а это означает, что либо должны разрабатываться специализированные рецепты (описывающие потенциальные ошибки или несоответствия актуальному окружению сборки), либо в подсистеме настройки системы непрерывной интеграции должен быть создан некоторый подходящий уровень абстракции для рецептов.
- *Хранение результатов обновления исходного кода/сборки/тестирования:* Хранение подробностей выполнения процессов обновления исходного кода (списка обновленных файлов, версии кода), сборки (сообщений с предупреждениями и ошибками) и тестирования (степени охвата исходного кода, производительности, степени использования памяти) для последующего анализа может оказаться желательным. Эти подробности могут быть использованы для ответов на вопросы, возникающие при тестировании программного обеспечения для различных архитектур (изменилась ли в значительной степени производительность программного обеспечения для какой-либо архитектуры после последнего обновления?) или в различные периоды времени (изменилась ли значительно степень охвата исходного кода в течение последнего месяца?) Как и в случае рецептов сборки, механизмы и типы данных, используемые на этом этапе, обычно специфичны для платформы или системы сборки.
- *Выпуск пакетов:* В результате сборки могут формироваться бинарные пакеты или другие продукты, которые должны быть доступны извне. Например, разработчики, не имеющие прямого доступа к машине для сборки, могут изъять желание участвовать в тестировании последней сборки программного обеспечения для определенной архитектуры; для этого система непрерывной интеграции должна иметь возможность передать собранные программные продукты в центральный репозиторий.
- *Сборки для множества архитектур:* Так как одной из задач непрерывной интеграции является сборка программного обеспечения для множества архитектур с целью тестирования кроссплатформенных функций, системам непрерывной интеграции может потребоваться отслеживать архитектуры каждой машины для сборки и связывать сборку и ее результаты с каждым клиентом.
- *Управление ресурсами:* Если этап сборки требует ресурсов, система непрерывной интеграции может потребовать использовать условия при сборке. Например, процесс сборки может ожидать отсутствия других процессов сборки или пользователей в системе или задерживаться до момента достижения определенных показателей загрузки центрального процессора и использования оперативной памяти.
- *Координация внешних ресурсов:* Процесс тестирования интеграции может зависеть от таких нелокальных ресурсов, как дополнительная база данных или удаленный веб-сервис. Таким образом, система непрерывной интеграции должна координировать работу процессов сборки на множестве машин для организации доступа к этим ресурсам.
- *Отчеты о выполнении действий:* При длительных процессах сборки регулярная отправка отчетов может быть также важна. Если пользователя в первую очередь интересуют результаты выполнения процесса сборки и тестирования в течение первых 30 минут вместо 5 часов, то принудительное ожидание завершения этого процесса для просмотра результатов приведет к потере времени.

Высокоуровневое представление этих возможных компонентов системы непрерывной интеграции показано на [Рисунке 9.1](#). Программное обеспечение непрерывной интеграции обычно реализует некоторое подмножество этих компонентов.

9.1.2. Внешние взаимодействия

Системам непрерывной интеграции также требуется осуществлять взаимодействия с другими системами. Существует несколько типов потенциальных взаимодействий:

- *Оповещения о сборке:* Выходные данные процессов сборки в общем случае должны быть переданы заинтересованным в них клиентам либо с помощью размещения информации (на веб-ресурсах, по RSS, RPC, и.т.д.), либо с помощью уведомления (посредством электронной почты, Twitter, PubSubHubbub, и.т.д.) Уведомление может относиться ко всем сборкам, неудачным сборкам или сборкам, не завершившимся в течение указанного периода времени.
- *Информация о сборке:* Иногда может потребоваться получение информации о сборке или передача собранных программных продуктов, что обычно реализуют с помощью механизма удаленного вызова процедур (RPC) или системы для получения больших объемов данных. Например, может возникнуть желание использовать внешнюю систему анализа для более глубокого или специфического анализа или создания отчета о степени охвата кода или производительности приложения. В дополнение к этому может быть создан внешний тестовый репозиторий с целью отслеживания успешных и неудачных результатов тестов вне системы непрерывной интеграции.
- *Запросы сборок:* Обработка внешних запросов сборок от пользователей или репозитория исходного кода также может оказаться необходимой. Большинство систем контроля версий имеют дополнительные точки вызова функций после модификации кода, с помощью которых может быть осуществлен удаленный вызов процедуры, например, инициирующей

сборку. Также пользователи могут вручную запрашивать сборки с помощью веб-интерфейса или удаленных вызовов других процедур.

- **Удаленное управление системой непрерывной интеграции:** В более общем случае весь процесс работы системы непрерывной интеграции может быть модифицирован в реальном времени с помощью более или менее хорошо проработанного интерфейса удаленных вызовов процедур. Либо расширения для вызова произвольных функций, либо формально спроектированный интерфейс могут потребоваться для выполнения сборок на заданных платформах, указания альтернативных веток исходного кода с целью осуществления сборки с различными патчами, а также для выполнения дополнительных сборок в зависимости от условий. Эти возможности полезны для поддержки систем с более обобщенным рабочим процессом, т.е. для разрешения модификации исходного кода только после прохождения измененным кодом всех тестов системы непрерывной интеграции или тестирования патчей в широком спектре систем перед финальной интеграцией. Ввиду использования различных систем регистрации ошибок, систем работы с патчами и других внешних систем, включение этой логики в состав системы непрерывной интеграции может не иметь смысла.

9.2. Архитектуры

В ходе разработки проектов Buildbot и CDash были выбраны диаметрально противоположные архитектуры, при этом были реализованы перекрывающиеся, но разделенные наборы возможностей. Ниже мы опишем эти наборы возможностей и обсудим то, насколько эти возможности проще или сложнее реализовать в зависимости от выбранной архитектуры.

9.2.1. Модель реализации: Buildbot

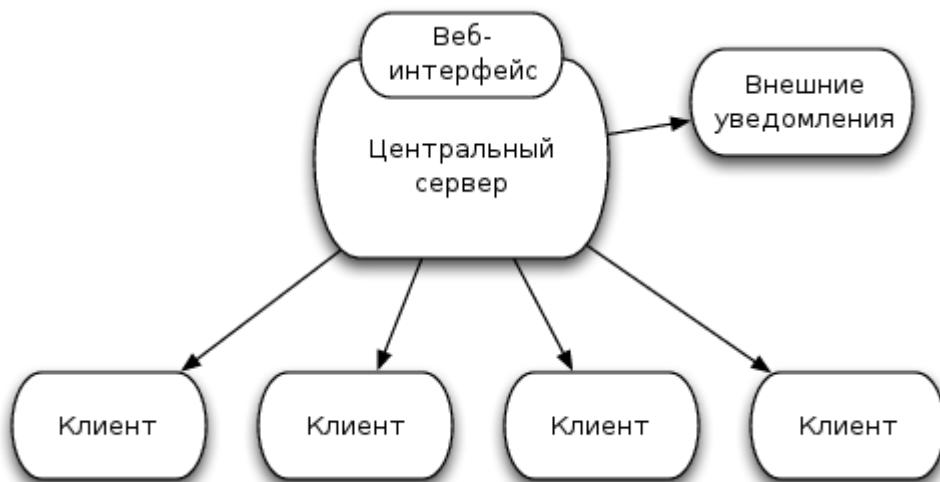


Рисунок 9.2. Архитектура системы Buildbot

Система Buildbot использует архитектуру ведущих/ведомых серверов, которая предусматривает единственный центральный сервер и множество управляемых им серверов для сборки. Удаленное исполнение команд осуществляется в полном соответствии со сценарием центрального сервера в реальном времени: центральный сервер отправляет команды для исполнения каждой из удаленных систем, которые начинают выполняться сразу же после завершения выполнения предыдущих команд. Планирование выполнения команд и запросы сборки не только координируются, но и полностью контролируются центральным сервером. Встроенной абстракции для рецептов сборки не существует, за исключением простейшей интеграции с системой контроля версий ("наш код в этом репозитории") и разделением команд для работы с директорией для сборки и команд для работы в директории для сборки. Специфические для операционных систем команды обычно задаются на этапе настройки.

Система Buildbot поддерживает постоянное соединение с каждой системой для сборки и осуществляет управление и координацию исполнения задач этими системами. Управление удаленными машинами посредством постоянного соединения добавляло сложностей при практической реализации и в течение длительного периода времени являлось источником ошибок. Длительная поддержка надежно функционирующих сетевых соединений не так проста и тестирование приложений, взаимодействующих с локальным графическим интерфейсом посредством сети, требует больших затрат труда. Особенно сложно работать с предупреждениями от операционной системы,

выводимыми в окнах сообщений. Однако, это постоянное соединение упрощает процессы координации и управления ресурсами, так как для выполнения задач ведомые машины находятся в полном распоряжении ведущей машины.

Тип непосредственного управления, применяемый в модели реализации Buildbot, делает централизованную координацию ресурсов для сборки достаточно простой. В системе Buildbot предусмотрены блокировки для центрального и ведомых серверов на центральном сервере, поэтому могут координироваться как глобальные сборки в рамках всей системы, так и локальные сборки в рамках ресурсов отдельных машин. Это обстоятельство делает систему Buildbot в большей степени пригодной для больших систем, на которых проводится тесты системной интеграции, т.е. тесты, в ходе которых осуществляется взаимодействие с базами данных или другими связанными с затратами ресурсов программными компонентами.

Следует учесть тот факт, что централизованная конфигурация обуславливает проблемы в случае использования распределенной модели. Каждый новый сервер для сборки должен быть явно задан при настройке центрального сервера, что делает невозможным динамическое подключение к центральному серверу новых серверов для сборки и их последующую эксплуатацию. Более того, так как каждый сервер для сборки находится под полным управлением центрального сервера, серверы сборки уязвимы для умышленных и случайных операций задания некорректных настроек: центральный сервер буквально осуществляет тотальный контроль клиента, контролируя также ограничения безопасности операционной системы клиента.

Одним из ограничений возможностей системы Buildbot является невозможность реализации простого способа возврата собранных программных продуктов на центральный сервер. Например, статистика охвата кода и бинарные пакеты хранятся на удаленном сервере для сборки, при этом не существует API для передачи их на центральный сервер с целью хранения и последующего распространения. Причины отсутствия данной возможности непонятны. Возможно, это последствие использования ограниченного набора распространяемых в составе Buildbot командных абстраций, которые главным образом направлены на удаленное выполнение команд серверами для сборки. Также, возможно это вызвано решением использовать соединение между центральным сервером и серверами сборки как систему контроля, а не механизм удаленного вызова процедур.

Другим последствием использования модели ведущих/ведомых серверов и ее ограниченного канала для взаимодействия является невозможность отправки отчетов о загрузке системы центральному серверу от ведомых серверов, поэтому центральный сервер не может избегать больших нагрузок на системы сборки.

Внешние уведомления о использовании центрального процессора в результате сборки в полной мере обрабатываются центральным сервером и новые службы уведомлений должны быть реализованы именно на стороне центрального сервера. Аналогично, новые запросы сборки должны отправляться напрямую центральному серверу.

9.2.2. Модель реализации: CDash



Рисунок 9.3: Архитектура CDash

В отличие от системы Buildbot, система CDash реализует модель сервера обработки отчетов. В этой модели сервер CDash выступает в качестве центрального репозитория для хранения информации о осуществленных удаленно сборках вместе с сопутствующими отчетами о неполадках при сборке и тестировании, результатах анализа охвата кода и использования памяти. Сборки осуществляются на удаленных клиентах по их собственному графику, после чего отправляются отчеты о сборке в формате XML. Отчеты о сборке могут отправляться как "официальными" клиентами, так и сторонними разработчиками или пользователями, выполняющими процесс сборки на своих машинах.

Реализация этой простой модели стала возможной благодаря тесной концептуальной интеграции между системой CDash и другими элементами инфраструктуры сборки программных компонентов компании Kitware: Cmake, системы настройки и сборки, CTest, системы тестирования и CPack, системы создания пакетов. Это программное обеспечение создает механизм, с помощью которого рецепты сборки, тестирования и создания пакетов могут быть реализованы на достаточно высоком уровне абстракции независимо от используемой операционной системы.

Процесс работы системы CDash с инициированием действий на стороне клиентов упрощает множество аспектов реализации процесса непрерывной интеграции на стороне клиента. Решение о начале сборки принимается клиентами, поэтому условия на стороне клиентов (время дня, высокая нагрузка, и.т.д.) могут быть приняты во внимание перед началом сборки. Клиенты могут появляться и исчезать по собственному желанию, что сводит на нет сложности проведения сторонних сборок и "сборок в облаке". Собранные программные продукты могут быть отправлены на центральный сервер с помощью простого механизма загрузки.

Однако, результатом использования модели обработки отчетов в CDash стало отсутствие многих полезных функций из системы Buildbot. Отсутствует централизованная координация ресурсов, причем эта функция не может быть просто реализована в распределенном окружении с непроверенными или ненадежными клиентами. Механизм отчетов о выполнении операций также не реализован: для реализации сервер должен позволять осуществлять последовательное обновление состояния сборки. И, конечно же, не существует способа для глобального запроса сборки с гарантией его выполнения анонимными клиентами в ответ на запрос - клиенты должны рассматриваться как ненадежные.

Не так давно в систему CDash были добавлены функции для работы системы сборки в облаке "@Home", в которой клиенты предоставляют ресурсы для сборки серверу CDash. Клиенты опрашивают сервер на наличие запросов на сборку, выполняют сборку в соответствии с запросами и возвращают результаты серверу. В текущей реализации (от октября 2010 года) сборки должны быть инициированы вручную на стороне сервера и клиенты должны быть соединены с сервером для предоставления своих услуг по сборке. Однако, эту модель достаточно просто усовершенствовать до более обобщенной модели планируемых сборок, в которой сборки запрашиваются автоматически сервером в случае доступности подходящего клиента. Система "@Home" по концепции очень похожа на систему Pony-Build, которая будет описана позднее.

9.2.3. Модель реализации: Jenkins

Jenkins является широко применяемой системой непрерывной интеграции, реализованной с использованием языка программирования Java; до начала 2011 года этот программный продукт был известен под названием Hudson. Данная система может выступать в роли как отдельной системы непрерывной интеграции, работая в локальной системе, так и координатора удаленных сборок, или даже пассивного сервера, принимающего информацию о сборках от удаленных серверов. Система использует преимущества стандарта JUnit XML для модульного тестирования и создания отчетов о степени охвата кода и интеграции отчетов от различных инструментов тестирования. Работа Jenkins была начата компанией Sun, но система широко применяется в различных компаниях и имеет надежное сообщество, сформированное вокруг ее открытого исходного кода.

Система Jenkins работает в гибридном режиме и по умолчанию использует режим выполнения сборок центральным сервером, при этом позволяя использовать множество методов для выполнения удаленной сборки, включающих в себя методы, инициирующие сборку как на стороне сервера, так и на стороне клиента. Однако, как и система Buildbot, данная система в первую очередь проектировалась для управления с помощью центрального сервера, но была адаптирована для поддержки широкого круга механизмов выполнения распределенных задач, включающих в себя механизмы управления виртуальными машинами.

Система Jenkins может управлять множеством удаленных машин с помощью соединения SSH, инициированного центральным сервером или клиентом с использованием технологии JNLP (Java Web Start). Это соединение является двухсторонним и позволяет осуществлять взаимодействие объектов и последовательную передачу данных.

В системе Jenkins применена сложная архитектура расширений, создающая абстракцию над функциями этого соединения, что позволило разработать множество сторонних расширений для поддержки возврата бинарных сборок и более важных выходных данных.

Для контролируемых центральным сервером задач система Jenkins предусматривает расширение "блокировок", позволяющее избежать параллельного выполнения этих задач, при этом по состоянию на январь 2011 года разработка этого расширения еще не завершена.

9.2.4. Модель реализации: Pony-Build



Рисунок 9.4: Архитектура системы Pony-Build

Pony-Build является экспериментальной децентрализованной системой непрерывной интеграции, разработанной с использованием языка Python. Она состоит из трех основных компонентов, изображенных на [Рисунке 9.4](#). Сервер обработки результатов выступает в роли централизованной базы данных, хранящей результаты, полученные от отдельных клиентов. Клиенты независимо хранят всю конфигурационную информацию и окружение сборки, совмещенное с легковесной клиентской библиотекой для облегчения доступа к репозиториям системы контроля версий, управления процессом сборки и передачи результатов серверу. Сервер обработки отчетов не является обязательным и поддерживает работу простого веб-интерфейса, предназначенного для вывода отчетов о результатах сборки и осуществления запросов новых сборок. В нашей реализации серверы обработки отчетов и результатов реализованы в рамках одного многопоточного процесса, но не имеют объединения на уровне API и без лишних сложностей могут быть запущены независимо друг от друга.

Эта простая модель усовершенствована с помощью множества различных точек вызова функций для работы с сетью и механизмов удаленного вызова процедур, способствующих отправке уведомлений о сборке и изменениях в системе, а также позволяющих воздействовать на процесс сборки. Например, вместо привязки уведомлений от репозитория исходного кода системы контроля версий напрямую к системе сборки, удаленные запросы направляются к системе обработки отчетов, которая передает их серверу обработки результатов. Аналогично, вместо формирования уведомлений о изменениях в новых сборках и отправки их с помощью электронной почты, системы мгновенных сообщений или других сервисов направляют серверу обработки отчетов, отправка уведомлений осуществляется с использованием активного протокола уведомлений PubSubHubbub (PuSH). Эта особенность позволяет получать широкому кругу различных приложений "интересующие" их уведомления (на данный момент набор уведомлений ограничен уведомлениями о новых сборках и неудачных сборках) с использованием точки вызова функции PuSH (PuSH webhook).

Преимущества этой разделенной модели весьма значительны:

- *Упрощение взаимодействия:* Основные компоненты архитектуры и протоколы с точками вызова функций могут быть реализованы чрезвычайно просто, при этом могут потребоваться только знания основ веб-программирования.
- *Упрощение модификации:* Реализация новых методов уведомлений или нового интерфейса сервера обработки отчетов чрезвычайно проста.
- *Поддержка множества языков программирования:* Так как различные компоненты вызывают друг друга с помощью точек вызова функций, которые поддерживаются множеством языков программирования, различные компоненты могут разрабатываться с применением различных языков программирования.
- *Простота тестирования:* Каждый компонент может быть полностью изолирован от системы и исследован, поэтому система может быть всесторонне протестирована.
- *Простота настройки:* Требования к программному обеспечению на стороне клиентов минимальны и заключаются в единственном файле библиотеки помимо самого интерпретатора Python.
- *Минимальная нагрузка на сервер:* Так как центральный сервер практически не управляет клиентами, разделенные клиенты могут выполнять задачи параллельно, не взаимодействуя с сервером и не загружая его работой за исключением моментов отправки отчетов.
- *Интеграция с системами контроля версий:* Настройка процесса сборки осуществляется в полной мере на стороне клиента, что позволяет использовать систему контроля версий.
- *Упрощение доступа к результатам:* Приложения, которые желают получать отчеты о сборках, могут быть разработаны с использованием любого языка программирования, позволяющего осуществлять удаленные вызовы процедур. Пользователям системы сборки может быть разрешен доступ к результатам сборок на сервере обработки отчетов на сетевом уровне или с помощью специализированного интерфейса сервера обработки отчетов. Клиентам для сборок доступ необходи́м исключительно для отправки результатов сборок серверу.

К сожалению, существует также множество серьезных недостатков, как и в случае модели системы CDash:

- *Сложность запроса сборок:* Эта сложность появляется в результате того, что клиенты для сборки абсолютно независимы от сервера обработки результатов. Клиенты могут опрашивать сервер обработки результатов для получения информации о запросах сборок, но такой подход обуславливает высокую нагрузку на систему и значительную задержку обработки запросов. Альтернативным вариантом является организация соединений для управления и контроля, чтобы сервер мог напрямую отправлять уведомления о запросах сборок клиентам. Данный подход обуславливает усложнение системы и сводит на нет достоинства разделения клиентов для сборки программного обеспечения.
- *Недостаточные возможности системы блокировки ресурсов:* Предоставление механизма удаленного вызова процедур для удержания и снятия блокировок ресурсов не является сложной задачей, гораздо сложнее применить политики использования ресурсов на стороне клиентов. Хотя такие системы непрерывной интеграции, как CDash и расчитывают на добровольное использование ресурсов на стороне клиента, клиенты могут столкнуться с непредвиденными сложностями, т.е. случаями непредусмотренного удержания блокировок. Реализация надежной распределенной системы блокировок сложна и добавляет нежелательной запутанности всей системе. Например, в общем случае для реализации блокировок ресурсов центрального сервера при работе с ненадежными клиентами система блокировок должна использовать политику работы с клиентами, которые заблокировали ресурс, но не сняли блокировку в ситуациях аварийного завершения работы или взаимной блокировки.
- *Недостаточные возможности отслеживания состояния системы в реальном времени:* Отслеживание состояния сборки в реальном времени и управление самим процессом сборки достаточно сложно реализуемы в системах без постоянного соединения. Одним важным преимуществом модели Buildbot над моделью с управлением на стороне клиентов является простота отслеживания промежуточного состояния длительных сборок, так как результаты сборки последовательно передаются через интерфейс центрального сервера. Более того, так как система Buildbot поддерживает управляющее соединение, в случае ошибки процесса сборки из-за некорректной настройки или ошибки в исходном коде, процесс сборки может быть прерван или отменен. Добавление такой возможности в систему Pony-Build, в которой сервер обработки результатов не имеет гарантий соединения с клиентами, потребует или постоянного опроса этого сервера клиентами, или реализации постоянно поддерживаемого соединения с клиентами.

Двумя другими аспектами реализации систем непрерывной интеграции, рассмотренными в ходе работы над Pony-Build являются предпочтительные методы реализации механизма рецептов сборки и управления доверенными ресурсами. Эти аспекты взаимосвязаны, так как при использовании рецептов сборки клиентами выполняется произвольный код.

9.2.5. Рецепты сборки

Рецепты сборки добавляют полезный уровень абстракции, особенно в случаях сборки кроссплатформенного программного обеспечения или использования мультиплатформенной системы сборки. Например, система CDash использует строгий тип рецептов сборки; большая часть, а возможно и все программное обеспечение, использующее для сборки систему CDash, подвергается обработке с помощью таких инструментов, как CMake, CTest и CPack, которые разрабатывались с учетом возможностей мультиплатформенных сборок. Это идеальная ситуация с точки зрения системы непрерывной интеграции, так как она может просто делегировать решение всех проблем сборочному инструментарию.

Однако, это справедливо не для всех языков программирования и окружений сборки. В экосистеме Python проводится стандартизация систем distutils и distutils2, предназначенных для сборки программных пакетов, но на данный момент не выработано стандарта, описывающего процесс подбора и выполнения тестов, а также объединения их результатов. Более того, во многие более сложные пакеты программ на языке Python добавлены специализированные логические функции для сборки, так как механизм расширений distutils позволяет выполнять произвольный код. Эта ситуация типична для всех инструментов сборки: наряду с существованием удачно стандартизированного набора команд для исполнения, всегда найдутся исключения и дополнения.

Реализация рецептов для сборки, тестирования и упаковки приводит к необходимости решения двух проблем: во-первых, рецепты должны создаваться независимо от платформы таким образом, чтобы один и тот же рецепт мог использоваться для сборки программного обеспечения на множестве систем; и во-вторых, они должны позволять осуществлять внесение изменений в процесс сборки с учетом особенностей программного обеспечения.

9.2.6. Управление доверенными ресурсами

Это третья проблема. Широкое использование рецептов сборки в системе непрерывной интеграции приводит к необходимости управления еще одним доверенным ресурсом: проверяться должно не только само программное обеспечение (так как клиенты системы выполняют произвольный код), но и рецепты сборки (так как они тоже должны иметь возможность выполнять произвольный код).

Эти вопросы управления доверенными ресурсами достаточно просто решаются в жестко контролируемом окружении, т.е. в компании, где клиенты сборки и система непрерывной интеграции являются частью внутреннего рабочего процесса. В других окружениях разработки, однако, заинтересованные третьи стороны могут предоставлять услуги сборки, например, проектам с открытым исходным кодом. Идеальным решением была бы поддержка включения стандартных рецептов для сборки в комплект поставки программного обеспечения на уровне сообществ, ведь сообщество Python уже задало это направление развития, создав систему distutils2. Альтернативным решением может быть разрешение использования рецептов сборки с цифровыми подписями, для того, чтобы надежные участники сообщества могли распространять подписанные рецепты сборки и клиенты системы непрерывной интеграции устанавливали, следует ли доверять рецептам сборки.

9.2.7. Выбор модели

По нашему опыту, свободное объединение удаленного вызова процедур и точек вызова функций в рамках модели системы непрерывной интеграции чрезвычайно просто реализуемо с учетом игнорирования всех требований тесной координации, которая обуславливает сложное со-

единение серверов. Простое выполнение удаленных проверок и сборок предполагает одинаковые требования к архитектуре в случаях как локальной, так и удаленной сборки; накопление информации о сборке (была ли сборка удачна/неудачна и.т.д.) в большей степени обуславливается требованиями к программному обеспечению на стороне клиента; а действия по отслеживанию информации на основе архитектуры и результатов обуславливаются теми же базовыми требованиями. Таким образом, простейшая система непрерывной интеграции может быть достаточно просто реализована с использованием модели обработки отчетов.

Мы также считаем модель свободного объединения компонентов очень гибкой и расширяемой. Добавление функций создания отчетов о новых типах результатов, механизмов уведомлений и рецептов сборки выполняется просто, так как компоненты четко разделены и достаточно независимы. Для разделенных компонентов четко установлены выполняемые ими задачи, также их просто тестировать и модифицировать.

Единственным сложным аспектом при реализации системы удаленных сборок в рамках аналогичной CDash модели свободного объединения компонентов является координация сборок: запуск и остановка процессов сборок, отчеты о выполняющихся сборках и координация блокировок ресурсов между несколькими клиентами технически более сложны в сравнении с остальной реализацией.

Достаточно просто сделать вывод о том, что модель свободного объединения компонентов "лучше" всех других, но очевидно, что это утверждение корректно только в том случае, когда координация сборок не требуется. Решение о выборе модели должно приниматься на основе требований проектов, для сборки которых будет использоваться система непрерывной интеграции.

9.3. Будущее систем непрерывной интеграции

При размышлении над устройством системы Pony-Build, мы сформулировали описание нескольких возможностей, которые хотелось бы увидеть в будущих выпусках систем непрерывной интеграции.

- *Независимый от языка программирования набор рецептов сборки:* На данный момент каждая система непрерывной интеграции идет по пути повторного изобретения колеса, реализуя свой собственный язык указания параметров сборки, что довольно забавно, ведь существует небольшое количество часто используемых систем сборки программного обеспечения и, вероятно, еще меньшее количество систем его тестирования. Несмотря на это, каждая система непрерывной интеграции использует новый и отличный от остальных способ задания команд для сборки и тестирования, которые должны быть выполнены. Фактически это обстоятельство является одной из причин существования такого множества в основном аналогичных систем непрерывной интеграции: каждое сообщество использует свой язык и создает свою систему настройки, привязанную к своим системам для сборки и тестирования, и основывающуюся на одних и тех же наборах низкоуровневых возможностей. Следовательно, создание предметно-ориентированного языка с возможностью представления параметров, используемых множеством наиболее часто применяемых наборов инструментов для сборки и тестирования позволит сгладить различия между системами непрерывной интеграции в значительной степени.
- *Применение стандартных форматов при формировании отчетов о сборке и тестировании:* Не существует четкого соглашения насчет того, какую именно информацию и в каком формате должны предоставлять системы сборки и тестирования. В случае создания формата или стандарта системам непрерывной интеграции будет значительно легче предоставлять как подробную, так и краткую информацию о сборках. Протокол Test Anywhere, TAP (от сообщества разработчиков языка Perl) и формат вывода результатов тестирования JUnit XML (от сообщества разработчиков языка Java) являются двумя интересными вариантами, с помощью которых можно закодировать информацию о количестве проведенных тестов, удачных и неудачных тестах и охвату кода для каждого из файлов.
- *Повышение уровня детализации и представления внутренних функций при создании отчетов:* Кроме того, удобным решением является предоставление хорошо документированного набора точек вызова функций для взаимодействия с системами настройки, компиляции и тестирования. Это позволило бы создать API (вместо стандартного формата), который могли бы использовать все системы непрерывной интеграции для извлечения подробной информации о сборках.

9.3.1. Заключительные размышления

Описанные выше системы непрерывной интеграции реализуют возможности, соответствующие их архитектуре, в то время, как система с гибридной архитектурой Jenkins начала свое развитие с

реализации архитектуры ведущего/ведомых серверов, в которую позднее были добавлены возможности, присущие архитектуре обработки отчетов со свободным объединением компонентов.

На основе этого наблюдения можно сделать вывод о том, что выбор архитектуры обуславливает функции системы. Конечно же, это не так. Скорее выбор архитектуры задает направление развития и реализации определенного набора функций. В ходе работы над системой Pony-Build мы были удивлены тем, настолько наш начальный выбор аналогичной системе CDash архитектуры обработки отчетов повлиял на будущие решения в области проектирования и реализации функций. Некоторые практические решения, такие, как уход от централизованной настройки и реализация подсистемы планирования в системе Pony-Build были приняты при учете наших условий эксплуатации данной системы: нам было необходимо обеспечить возможность динамического подключения удаленных клиентов для сборки, что сложно реализовать в случае использования системы Buildbot. Другие не реализованные нами в системе Pony-Build возможности, такие, как отчеты о состоянии сборки и централизованные блокировки ресурсов, были желательны, но очень сложны в реализации, для которой отсутствовали весомые аргументы.

Похожая логика может быть применена и к таким системам, как Buildbot, CDash и Jenkins. В каждом случае отсутствуют полезные возможности, вероятно, из-за несовместимости с выбранной архитектурой. Однако, после обсуждения с участниками сообществ Buildbot и CDash, а также чтения веб-сайта проекта Jenkins у нас сформировалось мнение о том, что необходимые функции были выбраны с самого начала, после чего началась разработка системы с использованием архитектуры, которая упрощает реализацию этих функций. Например, система CDash обслуживает сообщество с небольшим числом основных разработчиков, которые ведут разработку программного обеспечения, используя централизованную модель. Их главной задачей является поддержание работоспособности программного обеспечения на основных машинах, а вторичной - прием сообщений об ошибках от разбирающихся в технических тонкостях пользователей. В то же время, система Buildbot широко применяется в сложных окружениях с множеством клиентов, действия которых должны координироваться для доступа к разделяемым ресурсам. Более гибкий формат файлов настройки системы Buildbot с множеством параметров для планирования, изменения режима работы механизмов уведомлений и блокировок ресурсов подходит для этой цели лучше других. Наконец, система Jenkins нацелена на простоту использования и упрощение процесса непрерывной интеграции, используя для настройки графический интерфейс и параметры настройки для работы на локальном сервере.

Социальная составляющая процесса разработки программного обеспечения с открытым исходным кодом является другим фактором изменения соотношения между выбранной архитектурой и функциями: можете ли вы предположить, что разработчики выберут проекты с исходным кодом, основываясь на соответствии архитектуры проекта и его функций их условиям эксплуатации? Если это так, то их вклад в разработку будет направлен в общем случае на улучшение работы в условиях эксплуатации, которым соответствует проект. Таким образом, проекты могут быть ограничены определенным набором функций, так как участники процесса разработки сами проявляют инициативу и могут избегать архитектур, которые не предоставляют нужных им возможностей. Это утверждение было, конечно же, справедливо в нашем случае, когда мы решили реализовать новую систему Pony-Build вместо внесения изменений в систему Buildbot: архитектура системы Buildbot просто не подходила для сборки сотен тысяч пакетов.

Существующие системы непрерывной интеграции в основном создаются с использованием двух различных архитектур и обычно реализуют только часть из полезных функций. С развитием систем непрерывной интеграции и ростом числа их пользователей, мы ожидаем реализации дополнительных функций: однако, реализация этих функций может быть обусловлена начальным выбором архитектуры. Будет интересно проследить процесс развития систем этого типа.

9.3.2. Благодарности

Мы благодарим Greg Wilson, Brett Cannon, Eric Holscher, Jesse Noller и Victoria Laidler за интересные обсуждения систем непрерывной интеграции в общем и системы Pony-Build в частности. Некоторые студенты участвовали в развитии системы Pony-Build, а именно Jack Carlson, Fatima Cherkaoui, Max Laite и Khushboo Shakya.

10. Фреймворк Jitsi

Jitsi это приложение, которое позволяет пользователям совершать видео и голосовые звонки, совместно пользоваться своими рабочими столами, а также обмениваться файлами и сообщениями. Что еще более важно, оно позволяет это делать поверх ряда различных протоколов - от стандартных XMPP (Extensible Messaging и Presence Protocol) и SIP (Session Initiation Protocol) и до проприетарных, например, Yahoo! и Windows Live Messenger (MSN). Оно работает в Microsoft Windows, Apple Mac OS X, Linux и FreeBSD. Оно написано большей частью на языке Java, но в нем также есть части, написанные в нативном коде. В этой главе мы взглянем на архитектуру OSGi приложения Jitsi, рассмотрим, как она реализована и как управляет протоколами, а также оглянемся на то, что мы узнали при ее создании [1].

10.1. Разработка Jitsi

Тремя наиболее важными ограничениями, которые мы должны были иметь в виду при разработке Jitsi (в то время имевшего название SIP Communicator), были поддержка многих протоколов, кроссплатформенные операции и удобство для разработчика.

С точки зрения разработчик поддержка многих протоколов сводится к наличию общего интерфейса для всех протоколов. Другими словами, когда пользователь отправляет сообщение, наш графический интерфейс пользователя должно всегда вызывать один тот же метод sendMessage независимо от того, будет ли текущий выбранный протокол фактически вызывать метод, который называется sendXmppMessage или sendSipMsg.

Тот факт, что большая часть нашего кода написана на языке Java, удовлетворяет, в значительной мере, нашему второму ограничению: кроссплатформенности операций. Тем не менее, есть вещи, которые в среде Java Runtime Environment (JRE) не поддерживаются или делаются не так, как нам это нужно, например, захват видео с веб-камеры. Поэтому нам требуется использовать DirectShow в Windows, QTKit в Mac OS X и Video for Linux [2] в Linux. Точно также, как и с протоколами, те части кода, которые управляют видео, не должны касаться деталей управления (они достаточно сложны для этого).

Наконец, удобство для разработчика означает, что разработчикам должно быть легко добавлять новые функции. Сегодня VoIP пользуются миллионами, причем тысячами различных способов; различные провайдеры и разработчики сервис-услуг придумывают различные варианты использования и предлагаю новые идеи, касающиеся новых возможностей. Мы должны быть уверены, что им будет легко использовать Jitsi так, как они хотят. Тот, кому нужно добавить что-то новое, должен прочитать и понять только те части проекта, которые требуется модифицировать или расширить. Кроме того, изменение, делаемое одним человеком, должно минимальным образом влиять на все остальные работающие части проекта.

Если подвести итог, то нам нужна была среда, в которой различные части кода были бы относительно независимы друг от друга. Должна была быть возможность в зависимости от операционной системы легко менять некоторые части, запускать другие, например, протоколы, параллельно тем, что уже действуют, и иметь возможность полностью переписать любую из частей, причем так, чтобы остальной код работал без каких-либо изменений. Наконец, мы хотели иметь возможность легко включать и выключать отдельные части, а также иметь возможность загружать в наш список плагины через Интернет.

Мы кратко записали требования к нашему собственному фреймворку, но вскоре отказались от этой идеи. Нам не терпелось как можно скорее начать писать код для VoIP и IM и нам казалось, что не интересно тратить пару месяцев на фреймворк для плагинов. Кто-то предложил технологию OSGi, и она, как оказалось, подходит идеально.

10.2. Jitsi и фреймворк OSGi

Об OSGi написаны целые книги, так что мы не собираемся рассказывать о том, для чего предназначен этот фреймворк. Вместо этого мы только объясним, что он дает нам, и то, как мы используем его в Jitsi.

Кроме всего прочего, OSGi является модульным. Возможности в приложениях OSGi разделены на отдельные сборки. Сборка OSGi представляет собой немного большим, чем обычные файлы JAR, которые используются для распространения Java-библиотек и приложений. Jitsi представляет собой коллекцию таких сборок. Имеется сборка, которая отвечает за подключение к Windows Live Messenger, другая, в которой реализован XMPP, еще сборка, которая обрабатывает GUI, и так далее. Все эти сборки работают вместе в среде, которая с нашим случае предоставляется при помощи Apache Felix — реализации OSGi с открытым исходным кодом.

Все эти модули должны работать вместе. Сборка графического пользовательского интерфейса должна посыпать сообщения через сборки, реализующие протоколы, которые, в свою очередь, должны запоминать их с помощью сборок обработки истории сообщений. Это то, для чего предназначены сервисы OSGi: они представляют собой часть сборки, которая видна всем остальным. Сервис OSGi чаще всего является группой интерфейсов Java, которые позволяют использовать специальные функциональные возможности, например, ведение журнала, отсылки сообщений по сети или получение списка последних вызовов. Классы, в которых фактически реализованы функциональные возможности, известны как реализация сервисов. Большинство из них носят название интерфейса сервиса, который они реализуют, с суффиксом "Impl" в конце названия (например, ConfigurationServiceImpl). Фреймворк OSGi позволяет разработчикам скрывать реализации сервисов и быть уверенным, что они никогда не будут видны снаружи сборки, в которой они находятся. Таким образом, другие сборки могут использовать их только через интерфейсы сервисов.

В большинстве сборок также есть активаторы. Активаторы являются простыми интерфейсами, в которых определены методы `start` и `stop`. Каждый раз, когда Felix в Jitsi загружает или удаляет сборку, он вызывает эти методы с тем, чтобы сборку можно было подготовить к работе или к завершению. Когда эти методы вызываются, Felix передает им параметр, называемый `BundleContext`. `BundleContext` предоставляет сборкам метод подключения к среде OSGi. Таким образом, они могут обнаружить, какой сервис OSGi они должны использовать, или зарегистрироваться сами (рис.10.1).

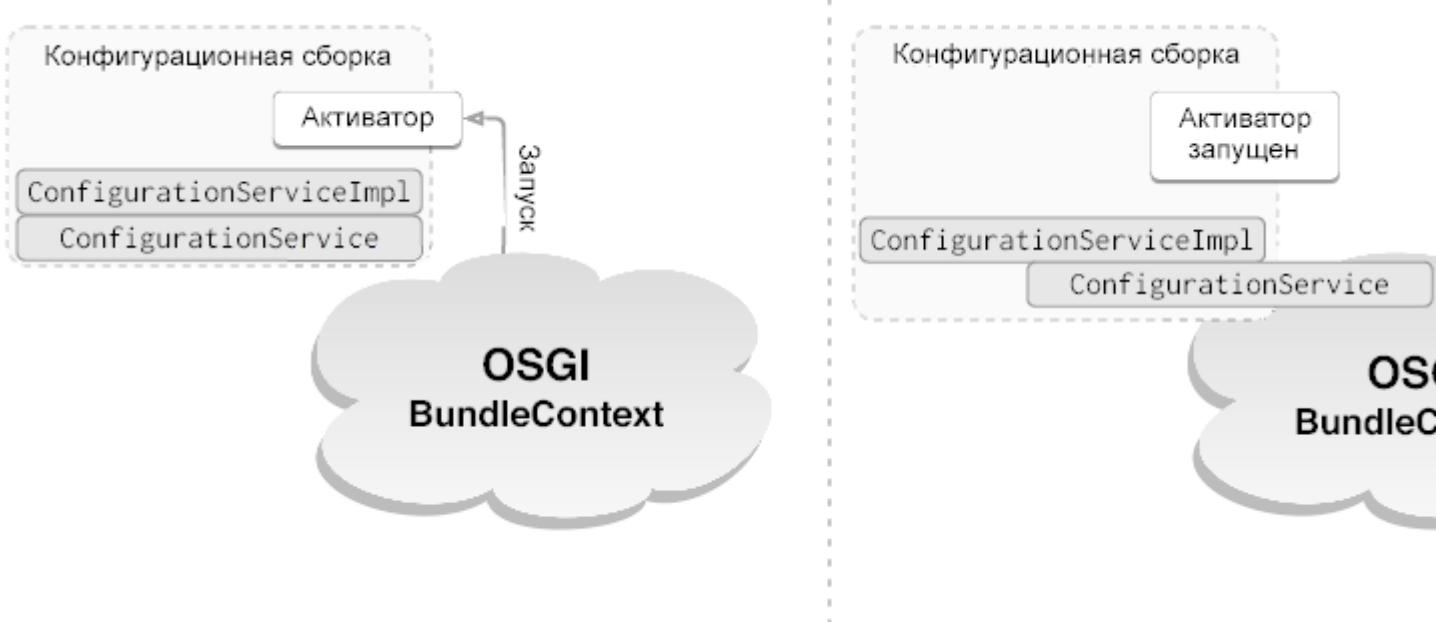


Рис.10.1: Активация сборки OSGi

Итак, давайте посмотрим, как это работает на практике. Представьте себе сервис, который реализует постоянное хранение и выборку свойств. В Jitsi это то, что мы называем ConfigurationService, и это выглядит следующим образом:

```
package net.java.sip.communicator.service.configuration;

public interface ConfigurationService
{
    public void setProperty(String propertyName, Object property);
    public Object getProperty(String propertyName);
}
```

Очень простая реализация ConfigurationService выглядит, например, следующим образом:

```
package net.java.sip.communicator.impl.configuration;

import java.util.*;
import net.java.sip.communicator.service.configuration.*;

public class ConfigurationServiceImpl implements ConfigurationService
{
    private final Properties properties = new Properties();

    public Object getProperty(String name)
    {
        return properties.get(name);
    }

    public void setProperty(String name, Object value)
    {
        properties.setProperty(name, value.toString());
    }
}
```

Обратите внимание, как сервис определен в пакете `net.java.sip.communicator.service`, а реализация - в `net.java.sip.communicator.impl`. Все сервисы и реализации разнесены в Jitsi по этим двум пакетам. OSGi позволяет, чтобы в сборках только некоторые пакеты были видны за предела-

ми своего собственного архива JAR, так что такое разделение облегчает сборкам экспортировать только пакеты сервисов и оставлять их реализации скрытыми.

Последнее, что нам нужно сделать с тем, чтобы можно было начать использовать нашу реализацию, это зарегистрировать ее в BundleContext и указать, что она является реализацией ConfigurationService. Вот как это происходит:

```
package net.java.sip.communicator.impl.configuration;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration.*;

public class ConfigActivator implements BundleActivator
{
    public void start(BundleContext bc) throws Exception
    {
        bc.registerService(ConfigurationService.class.getName(), // имя сервиса
                           new ConfigurationServiceImpl(), // реализация сервиса
                           null);
    }
}
```

После того, как класс ConfigurationServiceImpl зарегистрирован в BundleContext, другие сборки могут начать его использовать. Ниже приведен пример, показывающий, как некоторая случайнм образом выбранная сборка может использовать наш конфигурационный сервис:

```
package net.java.sip.communicator.plugin.randombundle;

import org.osgi.framework.*;
import net.java.sip.communicator.service.configuration.*;

public class RandomBundleActivator implements BundleActivator
{
    public void start(BundleContext bc) throws Exception
    {
        ServiceReference cRef = bc.getServiceReference(
            ConfigurationService.class.getName());
        configService = (ConfigurationService) bc.getService(cRef);

        // И это все! У нас есть ссылка на реализацию сервиса
        // и мы готовы запустить сохраненные свойства:
        configService.setProperty("propertyName", "PropertyValue");
    }
}
```

Еще раз обратите внимание на пакет. В net.java.sip.communicator.plugin мы сохраняем сборки, которые используют сервисы, определяемые другими сборками, но в которых ничего самостоятельно не экспортируется и не реализуется. Хорошим примером таких плагинов является конфигурационные формы: Они служат дополнениями к пользовательскому интерфейсу Jitsi, которые позволяют пользователям выполнять некоторые конкретные настройки приложения. Когда пользователи изменяют настройки, конфигурационные формы взаимодействуют с ConfigurationService или напрямую со сборками, ответственными за некоторую функциональную возможность. Впрочем, никакой из других сборок никогда не потребуется взаимодействовать с ними каким-либо другим образом (рис. 10.2).

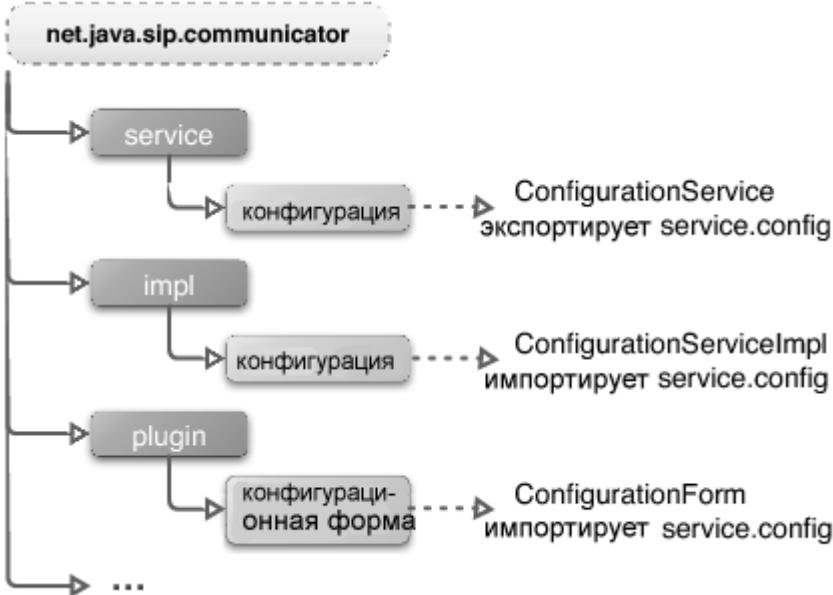


Рис.10.2: Структура сервиса

10.3. Собираем и запускаем сборку

Теперь, когда мы знаем, как в сборке писать код, наступил момент поговорить о формировании пакетов. Когда сборки работают, им надо указать три различные сущности среды OSGi: пакеты Java, которые они делают доступными для других (например, экспортные пакеты), пакеты, которые они хотели бы использовать из сборок (т.е. импортные пакеты), а также имя их класса BundleActivator. В сборках это делается при помощи манифеста JAR-файла, который используется при установке сборки.

Для сервиса ConfigurationService, который мы определили выше, файл манифеста может выглядеть следующим образом:

```

Bundle-Activator: net.java.sip.communicator.impl.configuration.ConfigActivator
Bundle-Name: Configuration Service Implementation
Bundle-Description: A bundle that offers configuration utilities
Bundle-Vendor: jitsi.org
Bundle-Version: 0.0.1
System-Bundle: yes
Import-Package: org.osgi.framework,
Export-Package: net.java.sip.communicator.service.configuration
  
```

После создания манифеста файла JAR, мы готовы создать саму сборку. В Jitsi мы используем Apache Ant, с помощью которого выполняются все задачи, необходимые для создания сборки. Для того, чтобы добавить сборку в процедуру создания Jitsi, вам нужно отредактировать файл build.xml в корневом каталоге проекта. JAR-файлы сборки создаются в конце файла build.xml там, где указываются задачи (цели) bundle-xxx. Для того, чтобы собрать наш конфигурационный сервис, нам понадобится следующее:

```

<target name="bundle-configuration">
  <jar destfile="${bundles.dest}/configuration.jar" manifest=
      "${src}/net/java/sip/communicator/impl/configuration/conf.manifest.mf" >

    <zipfileset dir="${dest}/net/java/sip/communicator/service/configuration"
                prefix="net/java/sip/communicator/service/configuration"/>
    <zipfileset dir="${dest}/net/java/sip/communicator/impl/configuration"
                prefix="net/java/sip/communicator/impl/configuration" />
  </jar>
</target>
  
```

Как вы можете видеть, задача в Ant просто создает JAR-файл используя для этого наш конфигурационный манифест и добавляет к нему пакеты `configuration` из иерархий `service` и `impl`. Теперь единственное, что нам нужно сделать, это чтобы Felix их загрузил.

Мы уже упоминали, что Jitsi является простым набором сборок OSGi. Когда пользователь запускает приложение, он, на самом деле, запускает Felix со списком сборок, которые необходимо загрузить. Вы можете найти этот список в нашем каталоге `lib`, внутри файла с именем `felix.client.run.properties`. Felix запускает сборки в том порядке, который определяется уровнями запуска: гарантируется, что сборки некоторого конкретного уровня будут запущены перед тем, как начнется загрузка следующего уровня. Хотя на примере кода, приведенного выше, это и не видно, настройки нашего конфигурационного сервиса сохраняются в файлах, поэтому нужно использовать наш файл `FileAccessService`, находящийся внутри файла `fileaccess.jar`. Так что мы проверяем, что `ConfigurationService` запускается после `FileAccessService`:

```
: : :
felix.auto.start.30= \
    reference:file:sc-bundles/fileaccess.jar

felix.auto.start.40= \
    reference:file:sc-bundles/configuration.jar \
    reference:file:sc-bundles/jmdnslib.jar \
    reference:file:sc-bundles/provdisc.jar \
: : :
```

Если вы посмотрите на файл `felix.client.run.properties`, вы увидите в начале список следующих пакетов:

```
org.osgi.framework.system.packages.extra= \
    apple.awt; \
    com.apple.cocoa.application; \
    com.apple.cocoa.foundation; \
    com.apple.eawt; \
: : :
```

В этом списке Felix-у указывается, какие пакеты, указываемые в системном пути `classpath`, должны быть доступными для сборок. Это означает, что пакеты, которые есть в этом списке, могут импортироваться сборками (т.е. могут быть добавлены в их заголовок манифеста *Import-Package*) без какого-либо явного указания на экспорт в любых других сборках. В списке, в основном, указываются пакеты, которые относятся к частям JRE, связанными с определенными особенностями ОС, и разработчикам Jitsi редко требуется добавлять туда новые пакеты; в большинстве случаев пакеты делаются доступными сразу целыми сборками.

10.4. Сервис провайдера протоколов

`ProtocolProviderService` в Jitsi определяет то, как будут себя вести все реализации протоколов. Это интерфейс, который в других сборках (например, в пользовательском интерфейсе) будет использоваться в случаях, когда нужно отправлять и получать сообщения, совершать звонки и обмениваться файлами через сети, к которым Jitsi подключен.

Все интерфейсы сервисов протоколов можно найти в пакете `net.java.sip.communicator.service.protocol`. Есть много реализаций сервиса, по одной для каждого поддерживаемого протокола, и все они хранятся в `net.java.sip.communicator.impl.protocol.protocol_name`.

Давайте начнем с каталогом `service.protocol`. Наиболее интересным фрагментом является интерфейс `ProtocolProviderService`. Всякий раз, когда нужно выполнить задачу, связанную с использованием протоколов, нужно искать реализацию этого сервиса в `BundleContext`. Сервис и его реа-

лизации позволяют Jitsi подключаться к любой из поддерживаемых сетей, получать статус и подробности соединения, а главное — получать ссылки на классы, в которых реализованы конкретные коммуникационные задачи, например, чат и осуществление звонков.

10.4.1. Наборы операций

Как мы уже упоминали ранее, `ProtocolProviderService` необходим для единообразного использования различных коммуникационных протоколов и сведения их различий к минимуму. Хотя это, в частности, исключительно просто для тех функций, которые используются во всех протоколах, например, функция отправки сообщений, все становится гораздо сложнее для тех задач, которые поддерживаются только в некоторых протоколах. Иногда эти различия связаны с самим сервисом: например, большинство сервисов SIP не поддерживают списки контактов, хранящихся на сервере, хотя это относительно хорошо поддерживаемая функция во всех других протоколах. Другим хорошим примером являются сервисы MSN и AIM: в свое время ни в одном из них не было возможности оставлять сообщения абонентам, не находящимся в сети, хотя в остальных сервисах эта возможность была (с тех пор все изменилось).

Суть заключается в том, что в нашем `ProtocolProviderService` должен быть способ обработки этих различий с тем, чтобы другие сборки, например, графический интерфейс, действовали согласовано; нет смысла добавлять в AIM `contact` кнопку вызова, если, в действительности, нет возможности делать вызов.

В качестве спасательного средства используется `OperationSets` (рис. 10.3). Неудивительно, что есть набор операций и предлагается интерфейс, который сборки Jitsi используют для управления реализациами протоколов. Методы, которые вы можете найти в интерфейсе набора операций, является тем, что касается конкретных возможностей.

В `OperationSetBasicInstantMessaging`, например, содержатся методы для создания и отправки мгновенных сообщений и регистрации слушателей (`listeners`), которые позволяют Jitsi находить сообщения, которые он получает. В другом примере, `OperationSetPresence`, есть методы запроса статуса контактов в вашем списке и установки собственного статуса. Поэтому, когда графический интерфейс обновляет статус отображаемого контакта или отправляет сообщение для конкретного контакта, он сначала может запросить у соответствующего провайдера, поддерживаются ли контакты и передача сообщений. К методам, которые в `ProtocolProviderService` определены для этой цели, относятся следующие:

```
public Map<String, OperationSet> getSupportedOperationSets();
public <T extends OperationSet> T getOperationSet(Class<T> opsetClass);
```

Наборы `OperationSets` должны быть сконструированы таким образом, чтобы была маловероятной такая ситуация, когда новый протокол, который мы добавим, имеет поддержку только некоторых из операций, которые определены в `OperationSet`. Например, в некоторых протоколах не поддерживается хранение списков контактов на сервере даже в случае, если в них пользователям разрешается запрашивать статус друг друга. Поэтому вместо того, чтобы в `OperationSetPresence` объединять управление хранением данных и использование списка друзей, мы также определили набор `OperationSetPersistentPresence`, который используется только с протоколами, в которых контакты можно хранить в Интернете. С другой стороны, мы еще не сталкивались с протоколом, в котором можно только посыпать сообщения и нельзя их получать, поэтому такие операции, как отправка и получение сообщений, можно смело объединять.

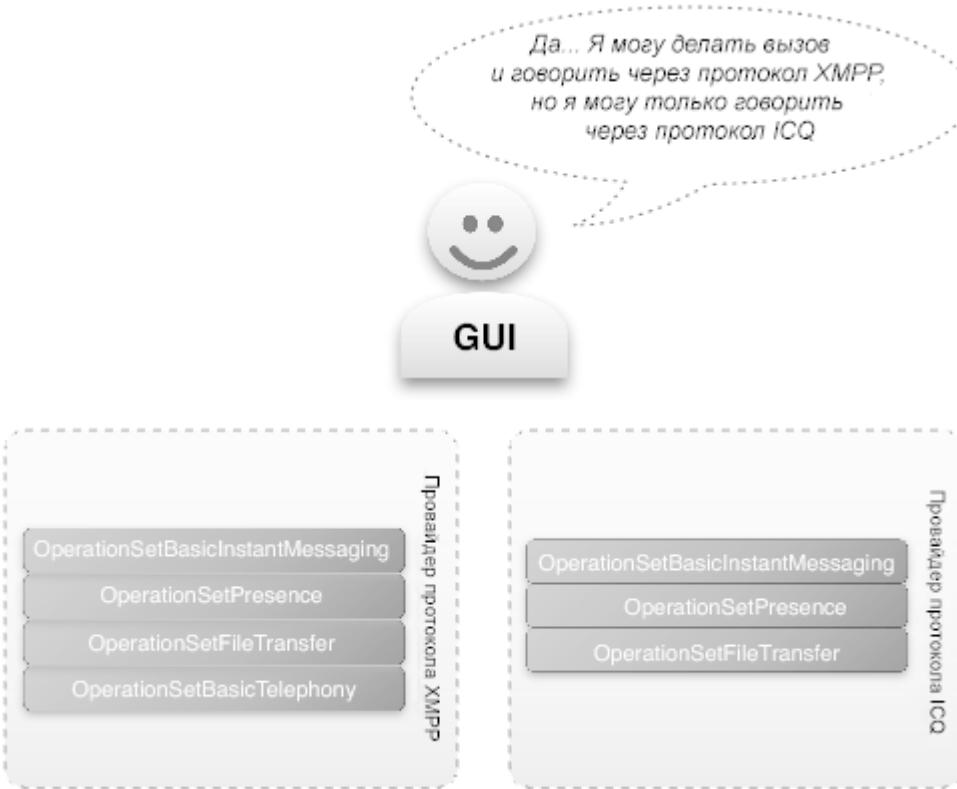


Рис.10.3: Наборы операций

10.4.2. Аккаунты, фабрики и экземпляры провайдеров

Важной характеристикой сервиса `ProtocolProviderService` является то, что один экземпляр соответствует ровно одному аккаунту протокола. Поэтому в любой заданный момент времени у вас в `BundleContext` есть столько возможных реализаций сервисов, сколько у вас есть аккаунтов, зарегистрированных пользователем.

В этот момент вам может стать интересно, кто создает и регистрирует провайдеров протоколов. В этом участвуют две сущности. Во-первых, имеется фабрика `ProtocolProviderFactory`. Это сервис, который позволяет другим сборкам получать отдельные экземпляры провайдеров, а затем регистрировать их в качестве сервисов. Для каждого протокола есть одна фабрика и каждая фабрика отвечает за создание провайдеров для конкретного протокола. Реализации фабрики хранятся вместе с остальными внутренними частями протокола. Например, для SIP у нас есть `net.java.sip.communicator.impl.protocol.sip.ProtocolProviderFactorySipImpl`.

Второй сущностью, участвующей в создании аккаунта, является визард протокола (`protocol wizard`). В отличие от фабрик, визарды отделены от остальной части реализации протокола, поскольку они связаны с графическим пользовательским интерфейсом. Визард, который позволяет пользователям создавать аккаунты для SIP, можно, например, найти в `net.java.sip.communicator.plugin.sipaccregwizz`.

10.5. Медиасервис

Когда работа со связью осуществляется в режиме реального времени поверх IP, имеется один важный аспект, о котором нужно иметь представление: протоколы, например, SIP и XMPP, хотя и признаются многими как наиболее распространенные протоколы VoIP, являются, на самом деле, не тем, что переносит голос и видео через Интернет. Эта задача выполняется с помощью протокола реального времени Real-time Transport Protocol (RTP). SIP и XMPP только отвечают за подготовку всего того, что требуется для RTP, например, определения адреса, куда должны отправляться пакеты RTP, форматов, в которых должны быть закодированы аудио и видео (например, кодек),

и т.д. На них также возлагается обязанность отслеживать присутствие пользователей, поддерживать собственное присутствие, выполнять телефонный вызов (делать звонок) и многое другое. Именно поэтому протоколы, например, SIP и XMPP, часто называют сигнальными протоколами.

Что это значит в контексте Jitsi? Ну, в первую очередь, это означает, что вы не отправитесь искать какой-нибудь код, в котором происходит обработка аудио или видео потоков в каком-нибудь из пакетов *jitsi sip* или *jabber*. Такой код находится в нашем медиасервисе MediaService. MediaService и ее реализация находятся в `net.java.sip.communicator.service.neomedia` и `net.java.sip.communicator.impl.neomedia`.

Почему "neomedia"?

« neo » в названии пакета NeoMedia указывает, что он заменяет аналогичный пакет, который мы использовали первоначально, и что затем нам потребовалось его полностью переписать. Это фактически одно из наших эмпирических правил, к которому мы пришли: вряд ли стоит тратить много времени на разработку приложения на 100% того, что потребуется в будущем. Просто нет возможности учесть все факторы, поэтому в любом случае вам обязательно позже потребуется делать изменения. Кроме того, вполне вероятно, что при кропотливом проектированию у вас повысится сложность решений, которые вам никогда не придется воспользоваться, поскольку сценарии, к которым вы будете готовы, никогда не возникнут.

В добавок к самому MediaService, есть два других интерфейса, которые особенно важны: MediaDevice и MediaStream.

10.5.1. Захват медиа, потоковая обработка и воспроизведение

Устройства MediaDevice представляют собой устройства захвата и воспроизведения медиапотока, которыми мы пользуемся во время разговора (рис. 10.4). Ваш микрофон и колонки, ваша гарнитура и ваша веб-камера являются примерами таких устройств MediaDevice, но они не единственные. Потоковая обработка на настольном компьютере и совместные звонки с захватом видео в Jitsi с вашего рабочего стола при использовании режима конференц-связи используют устройство AudioMixer для смешивания звука, который мы получаем от активных участников. Во всех случаях, устройства MediaDevice представляют собой лишь один тип MediaTure. То есть, все они могут быть только аудио или видео, но не обоих типов. Это означает, что если, например, у вас есть веб-камера со встроенным микрофоном, то Jitsi видит его как два устройства: одно, которое может только снимать видео, и еще одно, которое может захватывать только звук.

Однако, только устройств недостаточно для того, чтобы делать телефонные или видео вызовы. В дополнение к воспроизведению и захвату медиапотока, необходимо также иметь возможность отправлять его по сети. Это тот момент, когда используются интерфейсы медиапотоков MediaStream. Интерфейс MediaStream является тем, что подключается к устройству MediaDevice вашего собеседника. Здесь работа осуществляется с входящими и исходящими пакетами, которыми вы обмениваетесь с собеседником в течение вызова.

Точно также как и с устройствами, один поток может быть ответственен только за один тип MediaTure. Это означает, что в случае аудио/видео вызовов Jitsi должен создать два отдельных медиапотока, а затем подключить каждый к соответствующему аудио или видео устройству MediaDevice.

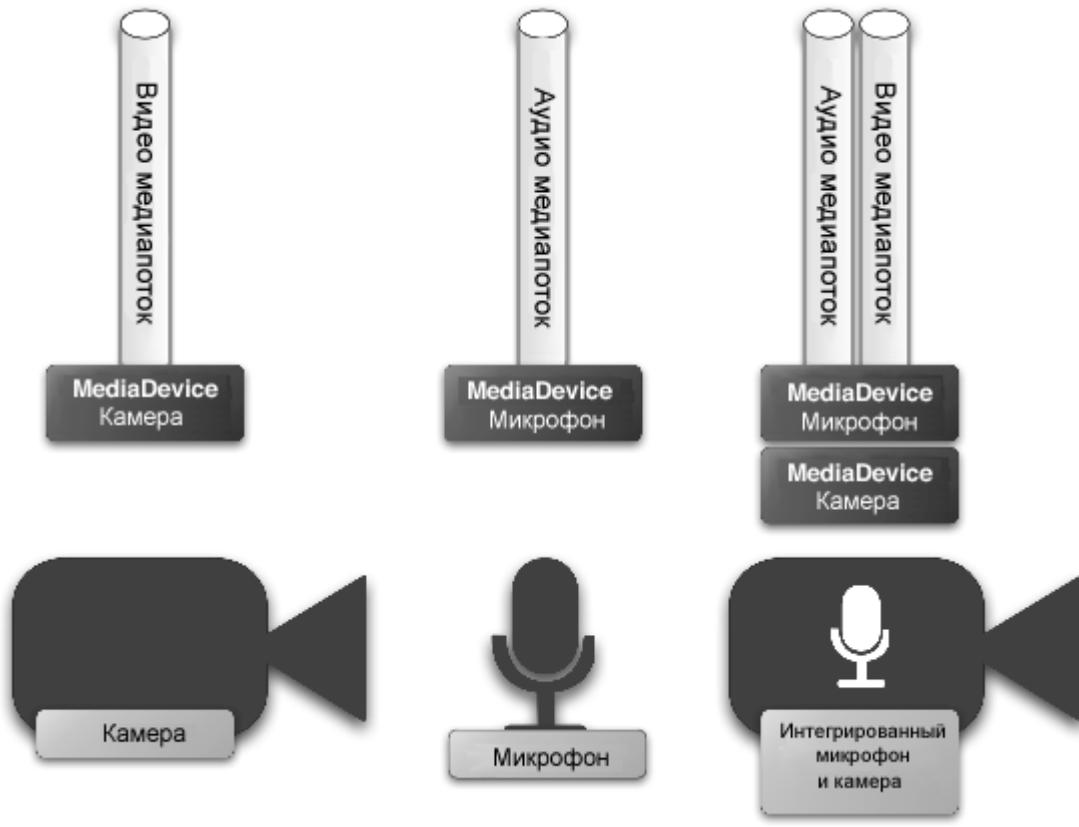


Рис.10.4: Медиапотоки для различных устройств

10.5.2. Кодеки

Другим важным понятием в потоковом медиа является то, что в MediaFormats, также известно как кодеки. По умолчанию большинство операционных систем позволяют записывать звук в формате 48 кГц PCM или в некотором подобном формате. Это то, что мы часто называем «необработанном аудио» («raw audio») и это тот вид аудио, который вы получаете в файлах WAV: отличное качество и огромный размер. Довольно непрактично пытаться передавать аудио через Интернет в формате PCM.

Это то, для чего предназначены кодеки: они позволяют вам представлять и транспортировать аудио и видео различным образом. Некоторые аудио кодеки, например, iLBC, 8KHz Speex или G.729, имеют низкие требования к пропускной способности, но звучат несколько приглушенно. Другие, например, широкополосные Speex и G.722, дают великолепное качество звука, но и требуют большей пропускной способности. Есть кодеки, которые пытаются обеспечить хорошее качество, сохраняя при этом требования к пропускной способности на разумном уровне. Хорошим примером такого кодека является популярный видео-кодек H.264. Компромисс здесь состоит в большом объеме расчетов, необходимых в процессе преобразования. Если вы используете Jitsi для видео вызовов в H.264, то вы обеспечите хорошее качество изображения и ваши требования к пропускной способности будут вполне разумными, но ваш процессор будет работать на максимуме.

Все это очень упрощенно, но идея в том, что выбор кодека всегда является выбором компромиссов. Вы либо жертвуете пропускной способностью, качеством, загрузкой процессора, либо их комбинацией. Людям, работающим с VoIP, редко нужно знать о кодеках больше.

10.5.3. Подключение к провайдерам протоколов

Протоколы в Jitsi, которые в настоящее время имеют поддержку аудио/видео, все используют наши медиасервисы Mediaservices одинаковым образом. Сначала они спрашивают MediaService об устройствах, доступных в системе:

```
public List<MediaDevice> getDevices(MediaType mediaType, MediaUseCase useCase);
```

Тип MediaType указывает, интересуют ли нас аудио или видео устройства. Параметр MediaUseCase в настоящее время рассматривается только в случае видео устройств. Это сообщает медиа сервису, хотим ли мы получить устройство, которые можно использовать при обычном вызове (MediaUseCase.CALL), в этом случае он возвращает список доступных веб-камер, или совместно используемые сессии настольного компьютера (MediaUseCase.DESKTOP), в этом случае он возвращает ссылки на настольные компьютеры пользователей.

Следующим шагом является получение списка форматов, доступных для конкретного устройства. Мы делаем это с помощью метода MediaDevice.getSupportedFormats:

```
public List<MediaFormat> getSupportedFormats();
```

Как только будет получен этот список, реализация протокола отправляет его другой стороне, которая вернет его подмножество с тем, чтобы указать, какие из них она поддерживает. Этот обмен также известен как модель Предложение/Ответ (Offer/Answer Model) и он часто использует протокол описания сеанса Session Description Protocol или некоторую другую его форму.

После обмена форматами и некоторыми номерами портов и IP-адресами, протоколы VoIP создают, настраивают и запускают медиапотоки MediaStreams. Грубо говоря, эта инициализация выполняется в следующих строках:

```
// Сначала создается коннектор потока, сообщающий медиасервису, какие сокеты
// должны использоваться, когда медиапоток транспортируется с помощью RTP, а
// управление потоком и статистика сообщений осуществляется с помощью RTCP
StreamConnector connector = new DefaultStreamConnector(rtpSocket, rtcpSocket);
MediaStream stream = mediaService.createMediaStream(connector, device, control);

// MediaStreamTarget указывает адрес и порт, которые наш собеседник использует
// для медиапотока. В различных протоколах VoIP есть свои собственные способы
// обмена этой информацией
stream.setTarget(target);

// Параметр MediaDirection сообщает потоку, откуда он поступает, куда он направляется,
// или и то и другое
stream.setDirection(direction);

// Затем мы задаем формат потока. Мы используем тот, который пришел первым в
// списке, вернувшись в ответе о согласовании сессии.
stream.setFormat(format);

// Наконец, мы готовы действительно начать получать медиапоток от нашего
// медиаустройства и направлять его в интернет
stream.start();
```

Теперь вы можете помахать в веб-камеру рукой, схватить микрофон и сказать: "Привет, мир!"

10.6. Сервис пользовательского интерфейса

До сих пор мы рассматривали те части Jitsi, которые имели дело с протоколами, отправкой и получением сообщений и осуществлении вызовов. Однако, Jitsi это, прежде всего, приложение, используемое реальными людьми и, поэтому, одним из наиболее важных аспектов является его пользовательский интерфейс. Большую часть времени пользовательский интерфейс использует

сервисы, в виде которых представлены все остальные сборки в Jitsi. Однако есть некоторые случаи, когда все происходит наоборот.

Первое, что приходит на ум, это - плагины. Плагинам в Jitsi часто нужно иметь возможность взаимодействовать с пользователем. Это означает, что они должны открывать, закрывать, перемещать или добавлять компоненты в существующие окна и панели в интерфейсе пользователя. Это то место, где в игру вступает наш сервис UIService. Он позволяет управлять основным окном в Jitsi, а также тем, как наши иконки будут позволять пользователям управлять приложением из dock-меню в Mac OS X или из области уведомлений в Windows.

Кроме простого использования списка контактов, плагины также могут выполнять дополнительные действия. Хорошим примером такого плагина является плагин, реализующий в Jitsi поддержку шифрования чата (OTR). Нашей сборке OTR необходимо зарегистрировать несколько графических компонентов в различных частях пользовательского интерфейса. Она добавляет кнопку замка в окно чата и подраздел в меню всех контактов, которое открывается правой кнопкой мыши.

Хорошо то, что он может делать все это с помощью вызова нескольких методов. В активаторе OSGi для сборки OTR, т. е. в OtrActivator, содержатся следующие строки:

```
Hashtable<String, String> filter = new Hashtable();

// Регистрируем элемент меню, открывающегося правой кнопкой.
filter.put(Container.CONTAINER_ID,
    Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID());

bundleContext.registerService(PluginComponent.class.getName(),
    new OtrMetaContactMenu(Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU),
    filter);

// Регистрируем элемент меню панели меню окна чата.
filter.put(Container.CONTAINER_ID,
    Container.CONTAINER_CHAT_MENU_BAR.getID());

bundleContext.registerService(PluginComponent.class.getName(),
    new OtrMetaContactMenu(Container.CONTAINER_CHAT_MENU_BAR),
    filter);
```

Как вы можете видеть, добавление компонентов в наш графический интерфейс пользователя просто сводится к регистрации сервисов OSGi. С другой стороны, наша реализация UIService выполняет поиск собственной реализации интерфейса PluginComponent. Всякий раз, когда она обнаруживает, что была зарегистрирована новая реализация, она получает ссылку на нее и добавляет ее в контейнер, указанный в фильтре сервиса OSGi.

Вот как это происходит в случае щелчка правой кнопкой мыши по пункту меню. В сборке пользовательского интерфейса имеется класс MetaContactRightButtonMenu, обрабатывающий щелчок правой кнопки мыши и в котором есть следующие строки:

```
// Поиск компонентов плагина, зарегистрированных через контекст сборки OSGI.
ServiceReference[] serRefs = null;

String osgiFilter = "("
    + Container.CONTAINER_ID
    + "==" + Container.CONTAINER_CONTACT_RIGHT_BUTTON_MENU.getID() + ")";

serRefs = GuiActivator.bundleContext.getServiceReferences(
    PluginComponent.class.getName(),
    osgiFilter);
// Проходим по всем плагинам, которые найдем, и добавляем их в меню.
for (int i = 0; i < serRefs.length; i++)
{
```

```

PluginComponent component = (PluginComponent) GuiActivator
    .bundleContext.getService(serRefs[i]);

component.setCurrentContact(metaContact);

if (component.getComponent() == null)
    continue;

this.add((Component) component.getComponent());
}

```

И это все, что нужно сделать. Большинство окон, которые вы видите в Jitsi, делают то же самое: Они ищут контекст сборки для сервисов, реализующих интерфейс PluginComponent, который имеет фильтр, указывающий, что их следует добавить в соответствующий контейнер. Плагины, как путешественники добирающиеся автостопом, держат таблички с названиями мест, куда они хотят добраться, превращая окна Jitsi в водителей, которые их подбирают.

10.7. Усвоенные уроки

Когда мы начали работу над SIP Communicator, одним из наиболее распространенных критических замечаний или вопросом, который мы слышали, был следующий: «Почему вы используете язык Java? Разве вы не знаете, что он медленный? Вы никогда не сможете получить достойное качество аудио/видео звонков!». Миф «язык Java - медленный» даже был повторен потенциальными пользователями в качестве причины того, почему они придерживаются Skype вместо Jitsi. Но первый урок, который мы усвоили из нашей работы над проектом, состоит в том, что с эффективностью на языке Java проблем не больше, чем их было с языком C++ или другими нативными альтернативами.

Мы не будем делать вид, что решение о выборе языка Java стало результатом тщательного анализа всех возможных вариантов. Нам просто нужен был простой способ создать нечто, что бы работало на Windows, и Linux, а Java и Java Media Framework, как нам показалось, предлагают один из относительно простых способов добиться этого.

На протяжении многих лет у нас было мало причин, чтобы сожалеть об этом решении. Совсем наоборот: даже хотя язык Java не сделал проект полностью прозрачным, он помогает обеспечивать переносимость и 90% кода в Communicator SIP не меняется от одной операционной системы к другой. Сюда относятся все реализации стека протоколов (например, SIP, XMPP, RTP, и т.д.), т. к. они достаточно сложны. Возможность не беспокоиться о специфике OS в таких частях кода оказывается весьма полезной.

Кроме того, популярность языка Java, оказалось очень важной при создании нашего сообщества. Авторы, как таковые, являются дефицитным ресурсом. Людям должно нравиться приложение, они должны найти время и мотивацию - все это трудно объединить вместе. Им не требуется учить новый язык и, таким образом, это - преимущество.

В отличие от большинства ожиданий, недостаточная скорость работы при использовании языка Java редко была причиной перехода на нативный код. Большинство решений, связанных с использованием нативного кода, обуславливались интеграцией с ОС и тем, насколько язык Java давал нам возможность получать доступ к утилитам конкретной OS. Ниже мы обсудим три наиболее важных области, в которых язык Java себя не оправдал.

10.7.1. Звук Java Sound и звук PortAudio

Java Sound является интерфейсом API, используемым в Java по умолчанию для записи и воспроизведения звука. Он является частью среды времени выполнения и, следовательно, работает на всех платформах, где есть виртуальная машина Java. В первые годы Jitsi, когда это был проект SIP

Communicator, использовался исключительно интерфейс JavaSound и из-за этого у нас было немало неудобств.

Прежде всего, интерфейс API не предоставил нам возможность выбирать, какие аудио устройства используются. Это большая проблема. Когда люди используют свои компьютеры для аудио и видео звонков, они для того, чтобы получить максимально возможное качество, часто применяют улучшенные гарнитуры USB или другие звуковые устройства. Когда в компьютере есть несколько таких устройств, JavaSound выбирает среди всех аудио устройств то, которое в операционной системе рассматривается как используемое по умолчанию, и это во многих случаях не всегда хорошо. Многие пользователи хотели бы сохранить настройки всех других приложений, работающих по умолчанию с их звуковой картой, так, чтобы, например, они могли продолжать прослушивать музыку через колонки. Еще более важно то, что во многих случаях для SIP Communicator лучше передавать аудио уведомление о вызове на одно устройство, а фактический звук вызова на другое, что позволяет пользователю, даже если он не находится перед компьютером, слышать сигнал вызова через колонки, а затем, когда он примет вызов, перейти к использованию гарнитуры.

Все это невозможно с Java Sound. Более того, в реализации для Linux используется OSS, что сегодня не рекомендуется использовать на большинстве дистрибутивов Linux.

Мы решили использовать альтернативную аудиосистему. Мы не хотели идти на компромисс с нашей мультиплатформенным решением и, если возможно, мы не хотели возиться со всем этим самостоятельно. Это та ситуация, когда чрезвычайно удобной оказалась система PortAudio [2].

Если Java самостоятельно не позволяет что-то делать, то следующим лучшим решением является использование кросс-платформенных проектов с открытым исходным кодом. Переход на проект PortAudio позволил нам реализовать поддержку лучшей настройки аудиопотоков так, как мы это описывали выше. Он также работает на Windows, Linux, Mac OS X, FreeBSD и других системах, для которых у нас не было времени создавать пакеты.

10.7.2. Захват и рендеринг видео

Видео для нас так же важно, как звук. Однако, это оказалось не так для создателей Java, поэтому по умолчанию в JRE нет интерфейса API, который позволяет выполнять захват или рендеринг видео. Некоторое время казалось, что таким API суждено было стать фреймворку Java Media Framework, до тех пор, пока фирма Sun не остановила его поддержку.

Естественно, мы начали искать альтернативу видео в стиле PortAudio, но на этот раз нам не так повезло. Сначала мы решили перейти на фреймворк LTI-CIVIL Кена Ларсона (Ken Larson) [3]. Это замечательный проект, и мы некоторое время его использовали [4]. Однако он оказался не слишком оптимальным при использовании в контексте коммуникаций реального времени.

Таким образом, мы пришли к выводу, что единственный способ обеспечить безупречную видеосвязь для Jitsi будет создание нативных грабберов и модулей рендеринга, которые мы должны были сделать самостоятельно. Это было не простое решение, поскольку оно подразумевало существенное увеличение сложности проекта и значительную нагрузку его поддержки, но у нас просто не было выбора: мы действительно хотели иметь качественную видеосвязь. И сейчас мы это делаем!

Наши нативные грабберы и модули рендеринга напрямую используют Video4Linux 2, QTKit и DirectShow/Direct3D на Linux, Mac OS X и Windows, соответственно.

10.7.3. Кодирование и декодирование видео

SIP Communicator, и, следовательно, Jitsi, с первых дней поддерживает видео-звонки. Это потому, что Java Media Framework позволял кодировать видео с использованием кодека H.263 и формата

176x144 (CIF). Те из вас, кто знает, как выглядит H.263 CIF, вероятно, сразу улыбнулись; мало кто из нас будет сегодня пользоваться приложением видео-чата в случае, если это все, что это приложение может предложить.

Для того чтобы обеспечить достойное качество, нам нужно было использовать другие библиотеки, например, FFmpeg. Кодирование видео, на самом деле, является одним из немногих мест, где производительности языка Java оказывается действительно недостаточно. То же самое касается других языков, о чем свидетельствует тот факт, что для того, чтобы обрабатывать видео наиболее эффективным способом, разработчики FFmpeg на самом деле в нескольких местах использовали Assembler.

10.7.4. Прочее

Есть много других мест, где мы решили, что для получения лучших результатов нам нужно перейти на нативный код. Одним из таких примеров являются уведомления системного трэя Systray с Growl - на Mac OS X и с libnotify — на Linux. К числу других примеров относятся базы данных запросов контактов из Microsoft Outlook и Apple Address Book, определяющие IP-адрес источника в зависимости от назначения, в которых использовались существующие реализации кодеков для Speex и G.722, позволяющие захватывать скриншоты рабочего стола и транслирующие символы char в коды нажатия клавиш.

Важно то, что всякий раз, когда нам нужно было выбрать нативное решение, мы могли это сделать и мы это делали. Это подводит нас к следующему: С тех пор, как мы запустили проект Jitsi, мы исправили, добавили, или даже полностью переписывали различные его части, поскольку мы хотели, чтобы они выглядели, воспринимались и работали лучше. Тем не менее, мы никогда не пожалели ни о чем, что не было первоначально написано правильно. Когда мы сомневались, мы просто выбрали один из имеющихся вариантов и использовали его. Мы могли бы подождать, пока мы не узнали лучше, что мы делаем, но если бы мы так поступали, то сегодня не было никакого проекта Jitsi.

10.8. Благодарности

Огромное спасибо Яне Стамчевой (Yana Stamcheva) за создание всех схем для этой главы.

Примечания

1. Чтобы непосредственно просматривать исходный код во время чтения главы, скачайте его с <http://jitsi.org/source>. Если вы используете Eclipse или NetBeans, вы можете скачать инструкцию о их настройке по ссылке <http://jitsi.org/eclipse> или <http://jitsi.org/netbeans>
2. <http://portaudio.com/>
3. <http://lti-civil.org/>
4. В действительности, у нас до сих пор есть этот вариант, который используется как неосновной.

Creative Commons. Перевод был сделан в соответствие с лицензией [Creative Commons](#). С русским вариантом лицензии можно ознакомиться [здесь](#).

Сноски

1. <http://llvm.org>
2. <http://clang.llvm.org>
3. Chris Lattner and Vikram Adve: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". Proc. 2004 International Symposium on Code Generation and Optimization (CGO'04), Mar 2004.

[На главную](#) -> [MyLDP](#) -> [Тематический каталог](#) ->

11. LLVM

В данной главе описываются некоторые технические решения, принятые в ходе работы над проектом LLVM¹, в рамках которого ведется разработка набора тесно связанных друг с другом низкоуровневых компонентов (ассемблеров, компиляторов, отладчиков, и.т.д.), которые в свою очередь разрабатываются с учетом совместимости с существующими и обычно использующимися в Unix-системах инструментами. Название "LLVM", изначально являющееся акронимом, на сегодняшний день стало брэндом. Хотя проект LLVM и предоставляет некоторые уникальные возможности, а также известен замечательными инструментами, поставляемыми в его составе (такими, как компилятор Clang², который работает с кодом на языках C/C++/Objective-C и имеет ряд преимуществ перед компилятором GCC), основным отличием LLVM от других компиляторов является его внутренняя архитектура.

С начала декабря 2000 года проект LLVM разрабатывался как набор библиотек для многоразового использования с хорошо проработанными интерфейсами [LA04]. В то время компиляторы с открытым исходным кодом разрабатывались как инструменты специального назначения, представленные монолитными исполняемыми файлами. Например, было очень сложно повторно использовать парсер из монолитного компилятора (к примеру, GCC) для проведения статического анализа или рефакторинга. Хотя интерпретируемые языки обычно и предоставляли возможность для включения их окружений времени исполнения и интерпретаторов в состав более сложных приложений, это окружение было представлено отдельным монолитным кодом, который мог либо включаться полностью, либо не включаться в состав приложений. Не было возможности повторного использования отдельных частей, разделение кода между реализациями языков программирования было крайне мало.

При проектировании самих компиляторов для популярных языков программирования, занимающиеся этим сообщества обычно были крайне поляризованы: реализации компиляторов обычно были представлены либо в традиционной форме статических компиляторов, таких, как GCC, Free Pascal и FreeBASIC, либо в форме компиляторов времени исполнения, представленной интерпретатором или компилятором Just-In-Time (JIT). Было очень сложно найти реализацию компилятора, предусматривающую работу в двух описанных выше режимах, а если она и находилась, то совместное использование кода между компиляторами разных типов было мало.

В течение последних десяти лет проекту LLVM удалось в значительной степени изменить этот порядок вещей. В настоящее время LLVM используется в качестве стандартной инфраструктуры для реализации компиляторов, работающих с большим количеством статически- и динамически-компилируемых языков (т.е. с семейством языков, поддерживаемых GCC, Java, .NET, Python, Ruby, Scheme, Haskell, D, а также с огромным количеством менее известных языков). Он также заменил собой множество компиляторов специального назначения, таких, как система специализации времени исполнения в стеке OpenGL компании Apple и библиотека обработки изображений в продукте After Effects компании Adobe. Наконец, LLVM также использовался при создании множества различных новых продуктов, наиболее известным из которых, скорее всего, является язык программирования для GPU OpenCL и окружение времени исполнения.

11.1 Краткое описание классической архитектуры компиляторов

Наиболее популярной архитектурой традиционных статических компиляторов (большинства компиляторов языка С) является архитектура трех фаз, главными элементами которой являются сис-

тема предварительной обработки, оптимизатор и система генерации кода (Рисунок 11.1). Система предварительной обработки производит разбор исходного кода, проверяет его на наличие ошибок, и строит специфическое для языка дерево абстрактного синтаксического анализа (Abstract Syntax Tree - AST) с целью представления исходного кода. В некоторых случаях дерево абстрактного синтаксического анализа конвертируется в новое представление для оптимизации, после чего оптимизатор и система генерации кода работают с кодом.



Рисунок 11.1: Три основных компонента компилятора на основе архитектуры трех фаз

Оптимизатор предназначен для проведения широкого круга преобразований с целью снижения времени исполнения кода, чаще всего путем исключения излишних расчетов, и обычно более или менее независим от языка программирования и целевой системы. Система генерации кода (также известная как кодогенератор) используется впоследствии для преобразования кода в набор инструкций целевой системы. В дополнение к генерации корректного кода, она должна генерировать качественный код, использующий нестандартные особенности поддерживаемой архитектуры. Стандартными механизмами системы генерации кода компилятора являются механизмы выбора инструкций, резервирования регистров и распределения инструкций.

Эта модель также отлично подходит к интерпретаторам и JIT-компиляторам. Виртуальная машина Java (JVM) также реализует данную модель, используя байткод при взаимодействии системы предварительной обработки и оптимизатора.

11.1.1 Последствия использования данной архитектуры

Наиболее важное достоинство данной архитектуры проявляется в том случае, когда от компилятора требуется поддержка нескольких языков программирования или нескольких целевых архитектур. Если компилятор использует стандартное представление кода при работе с оптимизатором, может быть разработана система предварительной обработки кода для любого языка программирования, для которого возможно преобразование кода в это представление, а также может быть разработана система генерации кода для любой целевой архитектуры, для которой может быть сгенерирован код на основе этого представления, как показано на Рисунке 11.2.



Рисунок 11.2: Многоцелевой компилятор

При использовании этой архитектуры доработка компилятора для поддержки нового языка (такого, как Algol или BASIC) требует реализации новой систем предварительной обработки кода, а оптимизатор и система генерации кода могут использоваться повторно. Если бы эти части не были разделены, реализация поддержки нового языка потребовала бы разработки нового компилятора,

поэтому для поддержки N целевых архитектур и M языков программирования потребовалось бы M*N компиляторов.

Другим преимуществом архитектуры трех фаз (являющимся прямым следствием работы компилятора с множеством целевых архитектур) является то обстоятельство, что компилятор используется большим количеством программистов, чем в том случае, если бы он поддерживал только один язык программирования и одну целевую архитектуру. Для проекта с открытым исходным кодом это означает, что вокруг проекта сформируется большее сообщество потенциальных разработчиков, что обычно приводит к расширению возможностей и улучшению компилятора. По этой причине компиляторы с открытым кодом, поддерживаемые множеством сообществ (такие, как GCC) обычно генерируют более качественно оптимизированный машинный код, нежели компиляторы, поддерживающие одним сообществом, такие, как FreePASCAL. Это утверждение не относится к проприетарным компиляторам, качество которых напрямую зависит от бюджета проекта. Например, компилятор ICC компании Intel широко известен благодаря высокому качеству генерируемого кода, хотя он и используется ограниченным кругом лиц.

Заключительным важным достоинством архитектуры трех фаз является то обстоятельство, что навыки, требуемые для разработки системы предварительной обработки кода отличаются от навыков, требуемых для разработки оптимизатора и системы генерации кода. Разделение этих систем упрощает работу участника, поддерживающего одну из систем предварительной обработки кода. Хотя это больше относится к социальным условиям, нежели к техническим, на самом деле это очень важно в особенности для проектов с открытым исходным кодом, разработчики которых хотят снизить барьер для вхождения новых разработчиков в проект настолько, насколько это возможно.

11.2. Существующие реализации языков программирования

Хотя достоинства архитектуры трех фаз бесспорны и хорошо описаны в книгах по разработке компиляторов, на практике данные архитектурные решения практически никогда не были реализованы в полной мере. Рассматривая реализации языков программирования (в момент начала работы над LLVM), вы можете обнаружить, что реализации таких языков, как Perl, Python, Ruby и Java не содержат общего кода. Более того, такие проекты, как Glasgow Haskell Compiler (GHC) и FreeBASIC позволяют генерировать машинный код для разных моделей центральных процессоров, но их реализации жестко привязаны к одному языку исходного кода, который они поддерживают. Существует также широкий круг технологий компиляции специального назначения, на основе которых реализуются JIT-компиляторы для работы с изображениями, регулярными выражениями, драйверами графических карт, а также работы в других областях, требующих интенсивного использования центрального процессора.

Следует отметить тот факт, что существуют как минимум три истории успеха, связанные с этой архитектурой, а первой из них являются виртуальные машины Java и .NET. Эти системы предоставляют JIT-компилятор, окружение времени исполнения и достаточно хорошо проработанный формат байткода. Это означает то, что любой язык, который может быть скомпилирован в формат байткода (и таких языков множество³), может использовать преимущества оптимизатора и JIT наряду с использованием окружения времени исполнения. При этом стоит учесть тот факт, что эти реализации ограничивают гибкость выбора окружений времени исполнения: обе реализации осуществляют эффективную JIT-компиляцию, сборку мусора и используют проработанную объектную модель. В итоге такой подход позволяет получить приемлемую производительность в случаях, когда компилируемые языки, такие, как язык C, не полностью подходят по критериям (в качестве примера можно привести проект LLJVM).

Второй историей успеха является скорее всего самый неудачный и при этом самый популярный подход повторного использования технологий компилятора: преобразование переданного исходного кода в исходный код языка C (или какого-либо другого языка) и обработка его с помощью

одного из существующих компиляторов С. Этот подход позволяет повторно использовать оптимизатор и генератор кода и является достаточно гибким решением, позволяющим контролировать окружение времени исполнения, а также данное решение позволяет участникам разработки системы предварительной обработки кода без лишних сложностей понять, реализовать и поддерживать ее. К сожалению, данный подход осложняет эффективную реализацию обработки исключений, позволяет использовать только ограниченные возможности отладки, замедляет процесс компиляции и может приводить к проблемам при работе с языками, требующими наличия гарантированных хвостовых вызовов (или других функций, не поддерживаемых языком С).

Завершающей список успешной реализацией данной модели является компилятор GCC⁴. GCC поддерживает множество систем предварительной обработки кода и разрабатывается активным и многочисленным сообществом. В течение длительного промежутка времени GCC был компилятором языка С с поддержкой различных целевых архитектур и с наличием поддержки нескольких дополнительных языков программирования, реализованной с использованием сложных приемов. По прошествии лет сообщество разработчиков GCC медленно реализовывало более совершенную архитектуру. В версии GCC 4.4 используется отдельное представление кода оптимизатором (известное как "Кортежи GIMPLE" - "GIMPLE Tuples"), которое в большей степени отделено от представления системы предварительной обработки кода, чем использующееся ранее. Также существуют системы предварительной обработки кода для языков Fortran и Ada, использующие необработанные деревья стандартного синтаксического анализа (AST).

Хотя эти продукты и были успешны, возможности их использования жестко ограничены, так как они проектировались как монолитные приложения. В качестве примера можно упомянуть о том, что компилятор GCC невозможно встроить в другие приложения, его невозможно использовать в качестве JIT-компилятора или интерпретатора, а также невозможно выделить и повторно использовать отдельные части GCC без задействования всего компилятора. Разработчикам, желающим использовать систему предварительной обработки кода для языка C++ из GCC в качестве инструмента для генерации документации, индексации кода, рефакторинга и статического анализа приходилось использовать GCC как монолитное приложение, выводящее интересующую их информацию в формате XML или разрабатывать расширения для использования стороннего кода в GCC.

Существует множество причин по которым отдельные части GCC не могут использоваться повторно в качестве библиотек, включающее в себя такие причины, как чрезмерное использование глобальных переменных, недостаточное использование инвариантов, некачественно проработанные структуры данных, распределенная кодовая база и использование макросов, которые затрудняют компиляцию кода для поддержки более чем одной комбинации из системы предварительной обработки кода и системы генерации кода. Наиболее сложные для исправления проблемы являются следствием недоработок в первоначальной архитектуре и возраста компилятора. В особенности GCC страдает от недоработок в распределении уровней и создании абстракций: система генерации кода получает доступ к дереву стандартного синтаксического анализа системы предварительной обработки кода для генерации отладочной информации, система предварительной обработки кода генерирует структуры данных для системы генерации кода и весь компилятор зависит от глобальных структур данных, изменяемых с помощью интерфейса командной строки.

Сноски

3. http://en.wikipedia.org/wiki/List_of_JVM_languages
4. Сейчас название расшифровывается как "GNU Compiler Collection".

11.3. Представление кода в LLVM: LLVM IR

После краткого экскурса в историю, давайте перейдем к подробному рассмотрению проекта LLVM: наиболее важным аспектом его архитектуры является использование промежуточного представления кода LLVM (LLVM Intermediate Representation - IR), являющегося формой для

представления кода компилятором. Представление LLVM IR спроектировано для проведения промежуточного анализа и преобразований, которые осуществляются системой оптимизации компилятора. Данное представление было разработано с учетом множества специфических задач, включающих в себя поддержку простых оптимизаций времени исполнения, кроссфункциональных и межпроцедурных оптимизаций, анализа всей программы, агрессивных преобразований реструктурирования, и.т.д. При этом наиболее важным является тот факт, что это представление само является языком с четко заданной семантикой. В качестве конкретного примера ниже приведено содержимое файла с расширением .ll:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Это представление LLVM IR соответствует следующему коду на языке C, иллюстрирующему два варианта сложения целых чисел:

```
unsigned add1(unsigned a, unsigned b) {
    return a+b;
}

// Не самый эффективный способ сложения двух чисел.
unsigned add2(unsigned a, unsigned b) {
    if (a == 0) return b;
    return add2(a-1, b+1);
}
```

Как вы видите в этом примере, представление LLVM IR является низкоуровневым виртуальным набором инструкций, подобным RISC. Как и в реальном наборе инструкций RISC, в нем поддерживаются линейные последовательности таких простых инструкций, как инструкции сложения, вычитания, сравнения и ветвления. Используются трехадресная форма инструкций, поэтому они принимают некоторое количество входных данных и помещают результат в отдельный регистр.⁵ Представление LLVM IR поддерживает метки и в общем случае выглядит как необычная форма ассемблерного кода.

В отличие от большинства наборов инструкций RISC, в LLVM применяется строгая типизация на основе простой системы типов (т.е., тип `i32` соответствует 32-битному целочисленному значению, тип `i32**` соответствует указателю на указатель на 32-битное целочисленное значение), а также некоторые системные особенности заменены абстракциями. Например, соглашение о вызове функций заменено абстракцией на основе инструкций `call` и `ret` с очевидными аргументами. Другим кардинальным отличием от машинного кода является использование представлением

LLVM IR бесконечного числа временных переменных с именами, начинающимися с символа %, вместо ограниченного набора именованных регистров.

Хотя представление LLVM IR и реализовано в форме языка, оно может быть представлено в виде трех изоморфных форм: в текстовом формате, описанном выше, в виде структур для хранения в памяти, исследуемых и изменяемых оптимизатором, и в виде эффективного и сжатого "биткода" для хранения на диске. Проект LLVM также предоставляет инструменты для преобразования хранящегося на диске представления из текстового в бинарный формат: `llvm-as` преобразует текстовый файл с расширением `.ll` в сжатый бинарный файл биткода с расширением `.bc`, а `llvm-dis` преобразует файл с расширением `.bc` в файл с расширением `.ll`.

Промежуточное представление кода интересно потому, что оно активно используется оптимизатором: в отличие от системы предварительной обработки кода и системы генерации кода компилятора, на работу оптимизатора не влияет ни исходный язык программирования, ни выбранная целевая архитектура. С другой стороны, оптимизатор должен хорошо обслуживать обе эти системы: он должен быть спроектирован таким образом, чтобы системе предварительной обработки кода было легче генерировать промежуточное представление, а также оставлять возможность для выполнения важных оптимизаций под заданные реальные архитектуры.

11.3.1. Разработка алгоритмов оптимизаций для LLVM IR

Для интуитивного понимания принципа работы оптимизаций полезно рассмотреть несколько примеров. Существует множество типов выполняемых компилятором оптимизаций, поэтому сложно обозначить принцип решения случайной задачи. При этом процесс выполнения большинства оптимизаций может быть разделен на три шага:

- Поиск шаблона для преобразования
- Проверка того, будет ли преобразование безопасным и корректным в данном случае.
- Выполнение преобразования, внесение изменения в код.

Наиболее тривиальной оптимизацией является оптимизация арифметических тождеств с помощью шаблонов, например: для любого целочисленного значения x , $x - x$ является 0, $x - 0$ является x , $(x * 2) - x$ является x . Первочередным является вопрос о том, как это будет выглядеть в представлении LLVM IR. Некоторые примеры приведены ниже:

```
:      :
%example1 = sub i32 %a, %a
:      :
%example2 = sub i32 %b, 0
:      :
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
:      :
```

Для данного типа "очевидных" преобразований LLVM предоставляет интерфейс упрощения инструкций, используемый различными преобразованиями более высоких уровней. Эти преобразования находятся в функции `SimplifySubInst` и выглядят следующим образом:

```
// X - 0 -> X
if (match(Op1, m_Zero()))
    return Op0;

// X - X -> 0
if (Op0 == Op1)
    return Constant::getNullValue(Op0->getType());
```

```
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
    return Op1;

...
return 0; // Совпадений не найдено, возвращается нулевое значение для указания на
          отсутствие преобразований.
```

В данном коде значения Op0 и Op1 привязаны к левому и правому операндам инструкции целочисленного вычитания (важно отметить, что эти обозначения не должны обязательно сохраняться для инструкций, работающих с числами с плавающей точкой IEEE!) LLVM реализован с использованием языка C++, который не является широко известным благодаря своим функциям для работы с шаблонами (в отличие от таких функциональных языков, как Objective Caml), но предоставляет чрезвычайно обобщенную систему шаблонов, которая позволяет нам реализовать подобные механизмы. Функция `match` и функции с префиксом `m_` позволяют нам осуществлять операции декларативного сопоставления шаблонов в коде представления LLVM IR. Например, функция `m_Specific` указывает на совпадение только тогда, когда значение выражения слева в инструкции умножения является таким же, как и Op1.

Во всех трех случаях используется сопоставление шаблонов и функция возвращает выражение для замены, если упрощение возможно, либо нулевой указатель, если упрощение невозможно. Данная функция (`SimplifyInstruction`) вызывается диспетчером, который обходит коды инструкций, выбирая соответствующую функцию обработки для каждого из кодов. Она вызывается из различных механизмов оптимизации. Простейший пример использования выглядит следующим образом:

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
    if (*Value *V = SimplifyInstruction(I))
        I->replaceAllUsesWith(V);
```

Данный код просто обходит все инструкции в блоке, проверяя возможность упрощения каждой из них. В случае возможности упрощения (функция `SimplifyInstruction` возвращает ненулевой указатель), используется метод `replaceAllUsesWith` для замены всех операций в коде на их упрощенную форму.

Сноски

5. В отличие от набора инструкций с двумя адресами, как в случае с архитектурой X86, где происходит деструктивное обновление содержимого исходного регистра, или систем с одним адресом, в которых принимается один операнд и производится работа с накопителем или с вершиной стека в соответствующих системах.

11.4. Реализация архитектуры трех фаз в LLVM

В компиляторе на основе LLVM система предварительной обработки кода осуществляет разбор, проверку и выявление ошибок в переданном коде, после чего преобразует разобранный код в представление LLVM IR (обычно, но не всегда путем формирования дерева стандартного синтаксического анализа (AST) и преобразования его в представление LLVM IR). Это представление может быть подвергнуто ряду операций по анализу и оптимизации для улучшения качества кода, после чего оно передается генератору кода для формирования машинного кода для необходимой архитектуры, как показано на Рисунке 11.3. Это достаточно линейная реализация архитектуры трех фаз, но простое объяснение демонстрирует всю мощь и гибкость, которой обладает архитектура LLVM благодаря использованию представления LLVM IR.

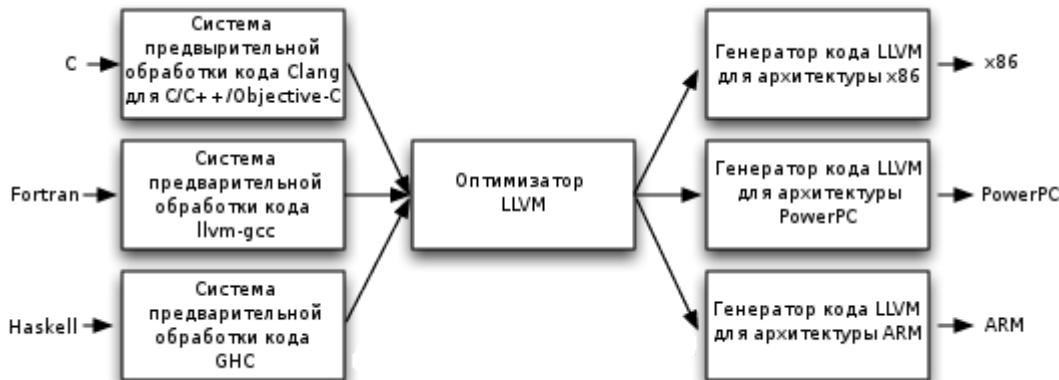


Рисунок 11.3: Реализация архитектуры трех фаз в LLVM

11.4.1 Представление LLVM IR является завершенным представлением кода

На самом деле представление LLVM IR является одновременно хорошо описанным и единственным интерфейсом для оптимизатора. Это означает, что все необходимые вам для разработки системы предварительной обработки кода знания заключаются в понимании устройства, принципов работы и используемых данных представления LLVM IR. Так как представление LLVM IR имеет текстовую форму, возможно и разумно создавать систему предварительной обработки кода, которая выводит представление LLVM IR в текстовой форме, после чего использует каналы Unix для передачи его выбранному оптимизатору и генератору кода.

Может показаться удивительным, но описанная выше возможность является оригинальной особенностью LLVM и одной из причин его успеха при работе в составе большого количества приложений. Даже очень успешный и обладающий относительно продуманной архитектурой компилятор GCC не имеет данной особенности: его промежуточное представление кода GIMPLE не является завершенным. В качестве простейшего примера следует упомянуть о том, что при генерации отладочной информации DWARF генератором кода из состава GCC, производится обход дерева исходного кода. Представление GIMPLE использует "кортежи" для обозначения операций в коде, но (по крайней мере в GCC 4.5) операнды все также обозначаются с помощью ссылок на дерево исходного кода.

В итоге разработчикам систем предварительной обработки кода необходимо обладать знаниями о том, как создается структуры дерева исходного кода GCC, а также о GIMPLE для их разработки. Генератор кода GCC обладает аналогичными недостатками, поэтому разработчикам приходится также разбираться в принципах его работы. Наконец, в GCC не предусмотрено возможности сохранения "завершенного представления кода" или способа чтения или записи представления GIMPLE (и соответствующих структур данных, формирующих представление кода) в текстовой форме. В результате эксперименты с GCC становятся относительно сложными, что ведет к уменьшению количества систем предварительной обработки кода.

11.4.2. LLVM является набором библиотек

Помимо представления LLVM IR, важным аспектом архитектуры LLVM является лежащий в его основе набор библиотек, а не монолитный компилятор с интерфейсом командной строки в случае GCC или сложная виртуальная машина в случае JVM или .NET. LLVM является инфраструктурой, набором полезных используемых в компиляторах технологий, которые могут быть использованы для решения специфических задач (таких, как включение компилятора языка С или оптимизатора в состав конвейера обработки спецэффектов). Эта возможность является одним из самых мощных и при этом плохо понимаемых архитектурных решений.

Давайте рассмотрим архитектуру оптимизатора, воспользовавшись примером: он читает представление LLVM IR, разделяет его на части, после чего генерирует представление LLVM IR, кото-

рое в теории может выполняться быстрее исходного. В LLVM (как и во многих других компиляторах) оптимизатор реализован в виде конвейера из разделенных фаз оптимизации, каждая из которых обрабатывает переданное представление и имеет возможность его модификации. Стандартными примерами оптимизаций являются: обработка inline-функций (при которой тела функций подставляются в места их вызовов), объединение выражений, оптимизация кода циклов, и.т.д. В зависимости от уровня оптимизаций выполняются различные фазы: например, при уровне `-O0` (без оптимизаций) компилятор Clang не использует фазы вообще, а при уровне `-O3` он использует серию из 67 фаз в ходе работы оптимизатора (в версии LLVM 2.8).

Алгоритм работы каждой фазы разрабатывается в виде отдельного класса C++, наследуемого (не напрямую) от класса `Pass`. Для большинства фаз используются отдельные файлы с расширением `.cpp` и их подкласс класса `Pass` объявляется в анонимном пространстве имен (что делает его полностью недоступным из файла объявления). Для того, чтобы было возможным использование фазы, у кода вне класса должна быть возможность использовать его, поэтому единственная функция (создания класса) экспортится из файла. Немного упрощенный пример алгоритма фазы для пояснения представлен ниже⁶:

```
namespace {
    class Hello : public FunctionPass {
        public:
            // Печать имен функций из представления LLVM IR, подвергающихся оптимизации.
            virtual bool runOnFunction(Function &F) {
                cerr << "Hello: " << F.getName() << "\n";
                return false;
            }
    };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

Как упоминалось ранее, оптимизатор LLVM позволяет использовать множество различных фаз, стили реализации алгоритмов каждой из которых идентичны. Эти алгоритмы компилируются в одни или несколько файлов с расширением `.o`, которые затем преобразуются в ряд статических библиотек (файлы с расширением `.a` в Unix-системах). Эти библиотеки предоставляют все виды функций для анализа и преобразования кода, а фазы по возможности объединяются: они могут использоваться отдельно или явно объявлять зависимости в том случае, если их работа зависит от результатов какого-либо анализа кода из другой фазы. При выполнении данной последовательности фаз система `PassManager` из состава LLVM использует явную информацию о зависимостях для удовлетворения этих зависимостей и оптимизации процесса выполнения фаз.

Библиотеки и абстрактные возможности замечательно проработаны, но они на самом деле не решают проблем. Интересно рассмотреть случай, когда кто-либо хочет создать инструмент, в котором возможно использование технологии компиляции, в частности, JIT-компилятор для языка обработки изображений. Разработчик этого JIT-компилятора сформулировал ряд условий: например, язык обработки изображений очень восприимчив к задержке при компиляции и имеет некоторые характерные свойства, поэтому необходимо произвести оптимизацию производительности.

Архитектура оптимизатора LLVM, основанная на библиотеках, позволяет нашему разработчику выбрать последовательность исполнения фаз, а также выбрать те фазы, которые необходимы в случае обработки изображений: если весь код находится в одной большой функции, нет смысла тратить время на обработку inline-функций. Если указатели используются крайне редко, не стоит беспокоиться об анализе ссылок и оптимизации использования памяти. Однако, несмотря на наши усилия, LLVM не сможет волшебным образом решить все задачи оптимизации! Так как система оптимизации разделена на модули и система `PassManager` не обладает информацией о внутренних алгоритмах фаз, у разработчика появляется возможность реализации своих собственных фаз оптимизации для данного языка, способных сгладить неточности работы оптимизатора LLVM в пла-

не явных специфических для данного языка возможностей оптимизации. На Рисунке 11.4 показан простой пример нашей гипотетической системы обработки изображений XYZ:

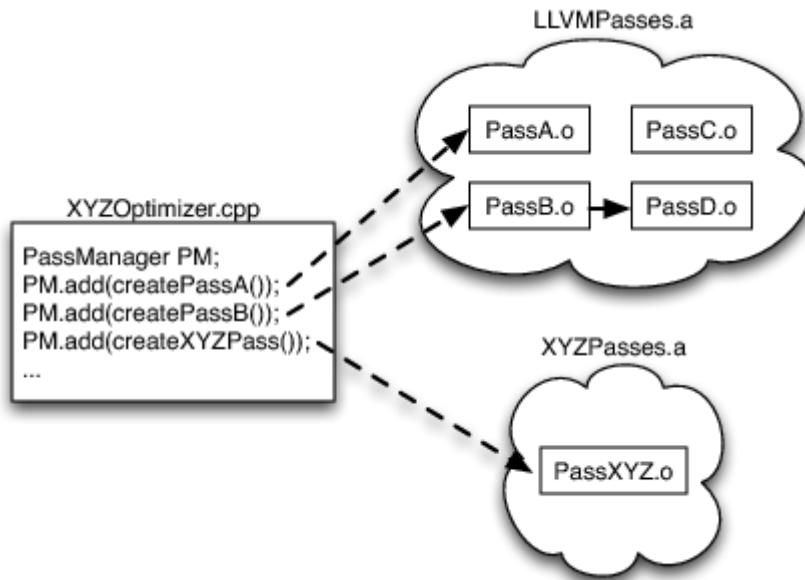


Рисунок 11.4: Гипотетическая система XYZ, использующая LLVM

Как только выбран набор оптимизаций (а также приняты подобные решения для генератора кода) компилятор для языка обработки изображений формируется в виде исполняемого файла или динамической библиотеки. Так как единственной ссылкой на алгоритм фазы оптимизации является простая функция `create`, объявляемая в каждом файле с расширением `.o`, а также оптимизаторы находятся в статических библиотеках с расширением `.a`, с приложением связывается код только тех фаз оптимизации, которые действительно используются, а не всех фаз оптимизации, доступных для LLVM. В нашем примере выше есть ссылки на код фаз `PassA` и `PassB`, поэтому они связываются с кодом приложения. Так как алгоритм фазы `PassB` использует алгоритм фазы `PassD` для проведения анализа, код фазы `PassD` также связывается с кодом приложения. Однако, так как фаза `PassC` (и множество других фаз оптимизации) не используется, код данной фазы не связывается с кодом приложения для обработки изображений.

В этой ситуации очевидна вся мощь архитектуры LLVM на основе библиотек. Это простое архитектурное решение позволяет LLVM предоставлять большое количество возможностей, некоторые из которых могут быть полезны только для определенного круга разработчиков, не лишая возможности использования библиотек тех, кому нужно выполнять только простейшие задачи. Напротив, в классических компиляторах оптимизаторы реализованы в виде тесно связанного кода большого объема, что осложняет его разделение на части, определение назначения его частей и ускорение его работы. В LLVM вы можете понять как работают отдельные оптимизаторы, не зная о том, как функционирует вся система.

Эта архитектура на основе библиотек также является причиной непонимания многими людьми принципов работы LLVM: библиотеки LLVM имеют множество возможностей, но они на самом деле ничего не делают сами по себе. Разработчик клиента для этих библиотек (например, компилятора языка C Clang) решает то, как они будут использоваться лучшим образом. Это тщательное разделение уровней, функций и внимание к разделению компонентов обуславливает возможность использования оптимизатора LLVM в таком широком диапазоне различных приложений для различных целей. Также тот факт, что LLVM позволяет использовать JIT-компиляцию не говорит о том, что каждый клиент использует ее.

Сноски

6. Для ознакомления с подробностями обратитесь к руководству "Writing an LLVM Pass manual" по адресу <http://llvm.org/docs/WritingAnLLVMPass.html>.

11.5. Архитектура многоцелевого генератора кода LLVM

Генератор кода LLVM преобразует код представления LLVM IR в машинный код для заданной целевой архитектуры. С другой стороны задачей генератора кода является формирование максимально качественного машинного кода для каждой из архитектур. В идеальном случае каждый генератор кода должен генерировать полностью отличающийся код для своей архитектуры, но с другой стороны генераторы кода для каждой целевой архитектуры решают аналогичные задачи. Например, каждая архитектура требует помещения значений в регистры и хотя каждая архитектура имеет отдельный файл со списком регистров, алгоритмы должны использоваться совместно там, где это возможно.

Аналогично подходу к реализации оптимизатора, генератор кода LLVM разделяет задачу по генерации кода на отдельные фазы - выбор инструкций, резервирование регистров, планирование использования регистров, оптимизация кода и генерация кода, а также предоставляет большое количество встроенных фаз, выполняющихся по умолчанию. Автор поддержки целевой архитектуры может выбрать стандартные фазы, заменить стандартные фазы или реализовать специфические для архитектуры фазы в случае необходимости. Например, система генерации кода для платформы x86 использует планирование распределения регистров, снижающее их использование, так как в данной архитектуре предусмотрено малое количество регистров, при этом система генерации кода для архитектуры PowerPC использует оптимизацию задержек, так как данная архитектура предусматривает большое количество регистров. Система генерации кода для архитектуры x86 использует специальную фазу генерации кода для обработки стека чисел с плавающей точкой x87, а система генерации кода для архитектуры ARM использует специальную фазу генерации кода для размещения наборов констант внутри функций по мере необходимости. Эта гибкость позволяет разработчикам систем генерации кода формировать качественный код без необходимости разработки с нуля всего генератора кода для их целевой архитектуры.

11.5.1. Файлы описания целевой архитектуры в LLVM

Техника "совмещения и сопоставления" позволяет разработчикам генераторов кода для архитектур выбирать необходимые для генерации кода действия и делает возможным повторное использование большого количества кода для разных архитектур. Из-за этого возникает еще одна сложность: каждый разделляемый между архитектурами компонент должен иметь возможность подстраиваться под свойства используемой архитектуры стандартным образом. Например, используемая в нескольких архитектурах система резервирования регистров должна иметь доступ к файлу описания регистров для каждой архитектуры и обладать информацией об условиях использования инструкций и их операндов совместно с регистрами. В LLVM используется решение, при котором для каждой целевой архитектуры создается описание на декларативном предметно-ориентированном языке программирования (набор файлов с расширением .td), обрабатываемое с помощью инструмента tblgen. Процесс генерации кода (упрощенный) для архитектуры x86 показан на Рисунке 11.5.

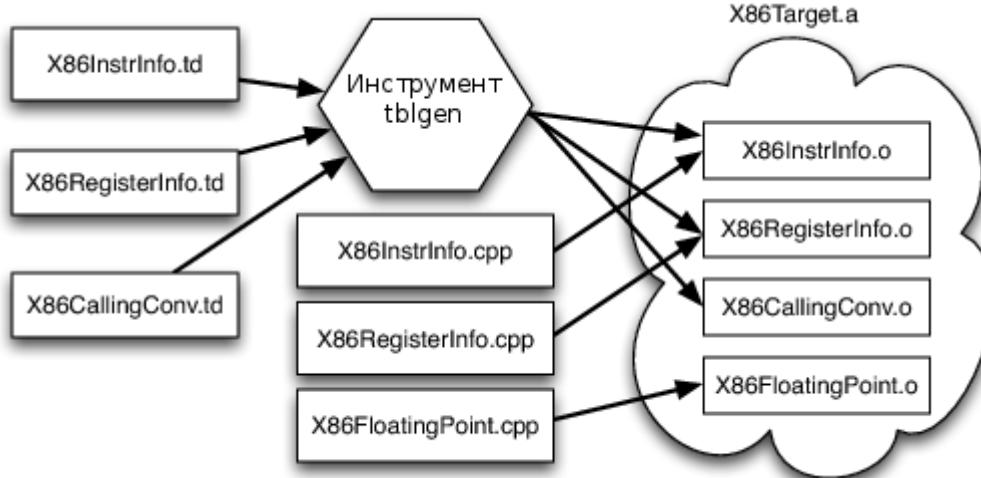


Рисунок 11.5: Упрощенный процесс генерации кода для целевой архитектуры x86

Различные подсистемы, поддерживаемые файлами с расширением .td позволяют разработчикам генераторов кода для разных архитектур последовательно обозначить особенности работы их архитектур. Например, система генерации кода для архитектуры x86 задает класс регистров, включающий в себя все 32-битные регистры и имеющий название "GR32" (в файлах с расширением .td специфичные для архитектуры объявления записываются заглавными буквами) следующим образом:

```
def GR32 : RegisterClass<i32>, 32,
  [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D] > { ... }
```

Это объявление сообщает о том, что регистры этого класса могут хранить 32-битные целочисленные значения ("i32"), предпочтительно их выравнивание по границам 32 бит, всего класс содержит 16 регистров (которые объявлены в одном из файлов с расширением .td), а также приводятся дополнительные данные для установления порядка резервирования регистров и других параметров. После добавления этого объявления в файл, специфические инструкции могут ссылаться на него, используя его в качестве операнда. Например, инструкция для работы с 32-битными регистрами описывается следующим образом:

```
let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
  (outs GR32:$dst), (ins GR32:$src),
  "not{1}\t$dst",
  [ (set GR32:$dst, (not GR32:$src)) ]>;
```

Эта описание сообщает о том, что NOT32r является инструкцией (используется класс `I` в `tblgen`), задается информация о кодировании (`0xF7, MRM2r`), задается объявленный для "вывода" 32-битный регистр с именем `$dst` и объявленный для "ввода" 32-битный регистр с именем `$src` (описанный выше класс регистров `GR32` устанавливает, какие регистры могут использоваться для операндов), задается синтаксис ассемблера для инструкции (используется объявление `()` для поддержки и синтаксиса AT&T и синтаксиса Intel), задается результат выполнения инструкции и шаблон, с которым должно совпадать объявление, в последней строке. Условие "let" в первой строке сообщает системе резервирования регистров о том, что для ввода и вывода данных должен быть зарезервирован один и тот же физический регистр.

Это объявление является очень точным описанием инструкции, поэтому стандартный код LLVM может быть сформирован с учетом полученной из него информации (с помощью инструмента `tblgen`). Одного этого объявления достаточно для системы выбора инструкций чтобы сформиро-

вать эту инструкцию на основе проверки совпадения с шаблоном данных из из IR-представления, передаваемых компилятору. Данное объявление также сообщает системе резервирования регистров о том, как работать с данной инструкцией, чего вполне достаточно для кодирования и декодирования инструкции в формат машинного кода, а также достаточно для поиска и вывода инструкции в текстовой форме. Эти возможности позволяют использовать систему генерации кода для архитектуры x86 в качестве отдельного ассемблера x86 (который может быть заменой ассемблера "gas" от GNU) и дизассемблеров на основе описания целевой архитектуры, а также кодировать инструкцию для использования ЛТ.

В дополнение к полезным функциям, наличие нескольких экземпляров данных, полученных из одного и того же источника полезно и для других целей. Данный подход делает практически невозможным рассогласование между ассемблером и дизассемблером в плане синтаксиса ассемблера и бинарного кода. Также это позволяет проводить простое тестирование: кодирование инструкций может быть проверено с помощью unit-тестирования без использования всего генератора кода.

Хотя в файлах с расширением .td и должно быть собрано столько полезной информации в удобной декларативной форме, сколько возможно, ее не достаточно. Напротив, от разработчиков систем генерации кода для разных архитектур требуется разработка кода на языке C++ для реализации различных функций и специфических для данной архитектуры фаз генерации исходного кода, которые могут понадобиться (таких, как x86FloatPoint.cpp для работы со стеком чисел с плавающей точкой). Так как в составе LLVM появляется поддержка новых архитектур, все более и более важным становится повышение количества архитектур, которые могут быть описаны с помощью файлов с расширением .td, поэтому и проводится работа по увеличению количества данных в этих файлах. Большим достоинством этого подхода является тот факт, что разрабатывать системы генерации кода для различных архитектур с использованием LLVM со временем становится все проще.

11.6. Интересные возможности, предоставляемые модульной архитектурой

Кроме того, что модульная архитектура является достаточно элегантным решением, она предоставляет клиентам библиотек LLVM некоторые интересные возможности. Эти возможности реализуются благодаря тому факту, что LLVM предоставляет функции, позволяя разработчику клиента выбирать большинство политик их использования.

11.6.1. Выбор момента и порядка выполнения каждой из фаз

Как упоминалось ранее, представление LLVM IR может быть эффективно преобразовано в бинарный формат, известный как биткод LLVM, а также из него. Так как представление LLVM IR является самодостаточным и процесс преобразования происходит без потерь, мы можем выполнить часть компиляции, сохранить данные на диск и после этого продолжить работу в будущем. Эта особенность позволяет реализовать ряд интересных возможностей, включающий в себя поддержку оптимизации времени связывания и времени установки, которые позволяют осуществлять генерацию кода после процесса компиляции.

Оптимизация времени связывания (Link-Time Optimization) решает проблему традиционной обработки компилятором одной единицы трансляции в каждый момент времени (т.е. файла исходного кода .c со всеми заголовками) и невозможности проведения оптимизаций (таких, как обработка inline-функций) в рамках нескольких файлов одновременно. Компиляторы на основе LLVM, такие, как Clang, поддерживают эту возможность с помощью аргументов командной строки -fllto или -O4. Эти аргументы сообщают компилятору о том, что нужно сформировать биткод LLVM и записать его в файл с расширением .o вместо записи объектного файла для данной архитектуры и задерживать генерацию кода до момента связывания, как показано на Рисунке 11.6.

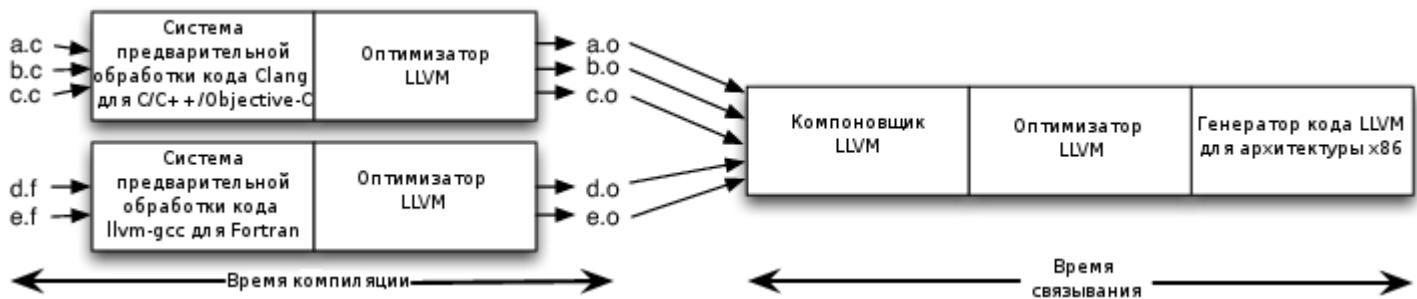


Рисунок 11.6: Оптимизация времени связывания

Подробное описание процесса данной оптимизации зависит от используемой операционной системы, но важным моментом является то, что компоновщик определяет, находится ли в файлах с расширением .o биткод LLVM или эти файлы являются объектными файлами для данной архитектуры. Когда установлено наличие биткода, компоновщик считывает содержимое файлов в память, производит связывание, после чего использует по отношению к коду оптимизатор LLVM. Так как теперь оптимизатор может обрабатывать гораздо больший объем кода, у него есть возможность преобразовывать inline-функции, устанавливать константы, проводить более агрессивное удаление фрагментов неиспользуемого кода и проводить другие оптимизации в рамках множества файлов. Хотя многие современные компиляторы поддерживают оптимизации времени связывания, большинство из них (т.е. GCC, Open64, компилятор Intel, и.т.д.) выполняют эти оптимизации с использованием ресурсоемкого и медленного процесса преобразования кода. В LLVM оптимизация времени связывания является естественным методом использования архитектуры системы и работает с различными исходными языками программирования (в отличие от множества других компиляторов), так как представление LLVM IR на самом деле не зависит от исходного языка программирования.

Оптимизация времени установки основана на идеи задержки процесса генерации кода на период до установки, превышающий по длительности период до связывания, как показано на Рисунке 11.7. Момент установки особенно интересен (в случаях, когда распространяются коробочные копии программного обеспечения, происходит его скачивание или установка на мобильное устройство, и.т.д.), так как в этот момент становятся известны особенности устройства, на котором программное обеспечение будет функционировать. В семействе устройств с архитектурой x86, например, существует множество чипов с различными характеристиками. Задерживая выбор инструкций, планирование и другие аспекты генерации кода, вы можете подстроиться под специфическое аппаратное обеспечение, на котором будет работать ваше приложение.

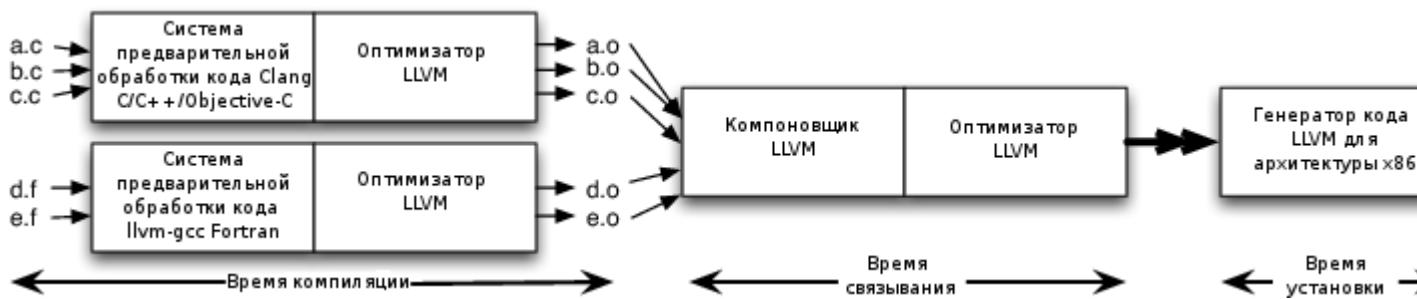


Рисунок 11.7: Оптимизация времени установки

11.6.2. Тестирование элементов оптимизатора

Компиляторы являются очень сложными программными продуктами, для которых важно качество, поэтому проведение тестирования просто необходимо. Например, после исправления ошибки, приводящей к краху оптимизатора, должно быть произведено тестирование на предмет регрессий для уверенности в том, что ошибка не проявляется снова. Традиционным способом тестирования является разработка программы на языке С и передача компилятору файла исходного кода с рас-

ширением .c для проверки корректности его работы и отсутствия аварийного завершения процесса компиляции. Этот подход, например, используется в наборе тестов компилятора GCC.

Недостатком этого подхода является то, что компилятор состоит из множества различных подсистем и даже различных фаз оптимизатора, каждая из которых имеет возможность изменить исходный код после некорректной работы предыдущей подсистемы. Если что-либо изменяется в системе предварительной обработки кода или на ранних стадиях работы оптимизатора, проверка с помощью данного подхода может оказаться невозможной.

Используя текстовую форму представления LLVM IR совместно с модульной архитектурой оптимизатора, набор тестов LLVM позволяет проводить тестирование на наличие регрессий отдельных компонентов, загружая представление LLVM IR с диска, подвергая его обработке с помощью одной фазы оптимизации и проверяя результат. Ниже приведен простой вариант теста, с помощью которого проводится проверка корректности работы алгоритма фазы оптимизации использования констант для инструкции сложения.

```
; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
    %A = add i32 4, 5
    ret i32 %A
; CHECK: @test()
; CHECK: ret i32 9
}
```

Строка с директивой `RUN` задает команду для выполнения: в данном случае используются инструменты с интерфейсом командной строки `opt` и `FileCheck`. Программа `opt` является всего лишь оберткой над менеджером фаз оптимизации LLVM, которая связана со всеми стандартными фазами (и может динамически подгружать дополнения с другими алгоритмами фаз) и позволяет использовать их с помощью интерфейса командной строки. Инструмент `FileCheck` проверяет, соответствуют ли данные, поступившие на стандартный ввод, серии описаний с использованием директив `CHECK`. В данном случае простейший тест проверяет корректность оптимизации операции сложения констант 4 и 5 с помощью инструкции `add` и получения в итоге значения 9 с помощью фазы оптимизации `constprop`.

Хотя этот пример и может выглядеть достаточно тривиальным, подобное тестирование сложно провести при помощи файлов исходного кода с расширением .c: обычно системы предварительной обработки кода производят действия с константами исходного кода во время его разбора, поэтому разработка кода, достигающего оптимизатора в первозданном виде, достаточно сложна и трудоемка. Так как мы можем загружать представление LLVM IR в виде текста и отправлять его для обработки с помощью интересующего алгоритма фазы оптимизации, после чего направлять вывод в другой текстовый файл, тестирование на наличие регрессий и корректность работы функций интересующих нас компонентов может осуществляться достаточно очевидно.

11.6.3. Автоматическое тестирование с помощью BugPoint

В случае обнаружения ошибки в компиляторе или в другом клиенте библиотек LLVM, первым шагом для ее устранения является создание условий для ее воспроизведения. Как только ошибка начинает воспроизводиться, следует минимизировать размер примера для ее воспроизведения и определить ту часть LLVM, в которой она проявляется, такую, как алгоритм фазы оптимизации. Хотя вы в конечном счете и научитесь это делать, данный процесс является скучным, неавтоматизированным и особенно сложным в случаях, когда компилятор генерирует некорректный код, но не завершается аварийно.

Инструмент BugPoint из состава LLVM⁷ использует преобразование в представление LLVM IR и модульную архитектуру для автоматизации этого процесса. Например, при передаче исходного

файла с расширением .ll или .bc вместе со списком фаз оптимизации, при использовании которых происходит крах оптимизатора, BugPoint сокращает объем кода до небольшого тестового примера и определяет, при выполнении какой из фаз оптимизации происходит крах. После этого он выводит сокращенный пример кода для тестирования и параметры командной строки для программы opt, позволяющие воспроизвести ошибку. Формирование результата путем сокращения объема кода и количества фаз оптимизации происходит с помощью техник, аналогичных отладке при помощи изменений ("delta debugging"). Так как BugPoint работает со структурой представления LLVM IR, не происходит траты времени впустую на генерацию некорректного представления и передачу его оптимизатору, как в случае использования стандартного инструмента "delta" с интерфейсом командной строки.

В более сложном случае при некорректной компиляции вы можете задать файл с кодом, информацию о генераторе кода, команды для передачи исполняемому файлу и образец вывода. BugPoint сначала определит, является ли источником проблемы оптимизатор или генератор кода, после чего будет постоянно разделять тестирование на две части: код будет передаваться "корректно работающему" компоненту и "некорректно работающему" компоненту. В ходе отправки все большего и большего объема кода некорректно работающему генератору кода, объем кода для воспроизведения неполадки сокращается.

BugPoint является очень простым инструментом, который сохранил бесконечное время, необходимое для тестирования, в течение всего периода существования LLVM. Ни один другой компилятор с открытым исходным кодом не имеет аналогичного инструмента, так как при его работе используется четко описанное промежуточное представление кода. Следует упомянуть о том, что BugPoint не является идеальным инструментом и мог бы быть значительно улучшен в случае полной переработки кода. Он был разработан в 2002 году и с того момента дорабатывался только тогда, когда кто-либо сталкивался с действительно сложной ошибкой, которую было невозможно отследить с помощью существующего инструмента. Тем не менее, со временем происходило расширение функций данного инструмента (таких, как отладка JIT-компилятора) в условиях отсутствия последовательной архитектуры и постоянного разработчика.

Сноски

7. <http://llvm.org/docs/Bugpoint.html>

11.7. Взгляд в прошлое и направления развития в будущем

Модульная архитектура LLVM изначально разрабатывалась не для достижения целей, описанных в данной главе. Она выступала в качестве механизма самозащиты: очевидно, что мы не могли сделать все правильно с первой попытки. Конвейер фаз, например, существует для изоляции фаз с целью их исключения после замены на более удачные реализации⁸.

Другим важным аспектом (и дискуссионной темой для разработчиков клиентов библиотек) является развитие LLVM и наше желание переосмыслить принятые ранее решения, внеся большие изменения в API без заботы об обратной совместимости. Кардинальные изменения представления LLVM IR, например, требуют изменения алгоритмов всех фаз оптимизации и приводят к значительным изменениям в API C++. Мы выполнили эти действия в нескольких случаях и, хотя это и нарушило работу клиентов, это также было важно для последующей поддержки высокого темпа развития. Для упрощения разработки сторонних клиентов (и для создания биндингов для других языков) мы предоставляем обертки на языке C для многих популярных API (которые остаются полностью стабильными), а также планируем, что новые версии LLVM смогут использовать файлы с расширениями .ll и .bc, созданные с помощью устаревших версий.

Смотря в будущее, мы хотели бы продолжить работу над модульной архитектурой LLVM и упрощением использования ее частей. Например, генератор кода все еще монолитен: на данный мо-

мент невозможно использовать его отдельные функции вне LLVM. Например, если вам необходим JIT-компилятор, но не требуется встроенного ассемблера, обработки исключений или вывода отладочной информации, у вас должна быть возможность сборки генератора кода без связывания с кодом для поддержки данных возможностей. Мы также постоянно улучшаем качество кода, генерируемого оптимизатором и генератором кода, добавляя дополнительные возможности в промежуточное представление IR для лучшей поддержки новых языков программирования и конструкций целевых архитектур, а также улучшаем поддержку оптимизаций, специфичных для языков программирования высокого уровня, в LLVM.

Проект LLVM продолжает развиваться и улучшается различными путями. Действительно захватывающе наблюдать множество способов использования LLVM в сторонних проектах и появление его в таких неожиданных контекстах, о которых даже не могли подумать проектировщики. Новый отладчик LLDB является хорошим примером этого: он использует системы разбора кода C/C++/Objective-C из проекта Clang для разбора выражений, использует LLVM JIT для преобразования их в код для целевой архитектуры, использует дизассемблеры LLVM, а также использует описания целевых архитектур из состава LLVM для обработки соглашений о вызовах и других вещей. Возможность повторного использования существующего кода позволяет разработчикам отладчиков сфокусироваться на логике работы отладчика вместо повторной реализации еще одной (предельно корректной) системы разбора кода C++.

Несмотря на успех, многое еще предстоит сделать и при этом постоянно существует риск превращения со временем LLVM в медленно развивающийся проект. Хотя не существует волшебного решения данной проблемы, я надеюсь, что продолжающееся исследование новых областей применения, желание переосмыслиния принятых ранее решений и изменение архитектуры с удалением устаревшего кода помогут в этом. Наконец, я хотел бы подчеркнуть, что целью разработки является не создание превосходного программного продукта, а его постоянное улучшение.

Сноски

8. Я всегда говорю, что подсистема в LLVM не является действительно качественно реализованной до того момента, пока не произведена ее повторная разработка хотя бы раз.

12. Архитектура Mercurial

Глава 12 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Mercurial — это современная распределенная система контроля версий (VCS), написанная главным образом на Python и частично на C (там, где критична производительность). В данной главе я расскажу о некоторых решениях, которые были приняты при написании алгоритмов и структур данных Mercurial. Для начала позвольте мне вкратце описать историю систем контроля версий, чтобы было понятна обстановка, в которой появился Mercurial.

12.1. Краткая история контроля версий

Несмотря на то, что данный текст главным образом посвящен архитектуре Mercurial, многие описываемые в нем концепции схожи с другими системами контроля версий. Для плодотворного рассказа о Mercurial я бы хотел описать некоторые такие концепции, применяемые в других системах. Для создания общей перспективы я также приведу краткую историю этих программ.

Системы контроля версий были изобретены для помощи разработчикам в совместной работе над проектами без необходимости ручного отслеживания изменений в файлах и передачи полных копий проекта. В данном тексте я буду вести речь не об исходном коде проекта, а вообще произвольном файловом дереве. Одной из главных функций программ контроля версий является передача изменений в такое дерево. Обычный цикл работы при этом выглядит следующим образом:

1. Получение самой последней версии структуры файлов от кого-то.
2. Работа над этой версией структуры, создание набора измененных файлов.
3. Публикация этих изменений, чтобы другие могли их использовать.

Первое действие, получение файловой структуры, называется *checkout*. Хранилище, из которого мы получаем и куда отправляем наши изменения, называется репозиторий, а результат *checkout*'а называется */i>рабочей директорией или рабочей копией*. Обновление рабочей копии последними версиями файлов из репозитория называется просто *апдейт*; иногда при этом требуется *слияние* (*merge*), то есть совмещение изменений от разных пользователей в одном файле. Команда **diff** позволяет просмотреть различия между двумя версиями структуры или файла, обычно при этом проверяются локальные (неопубликованные) изменения в вашей рабочей копии. Изменения публикуются при помощи команды *commit*, которая сохраняет изменения из рабочей директории в репозиторий.

12.1.1. Централизованный контроль версий

Первой системой контроля версий была Source Code Control System, SCCS, появившаяся в 1975 году. Она главным образом выполняла сохранение изменений между файлами кода в отдельные файлы, что было более эффективным, чем просто хранение копий, но не помогала в отправке этих изменений остальным участникам рабочего процесса.

В 1982 году ей на смену пришла Revision Control System, RCS, которая была более развитой и бесплатной альтернативой SCCS (и которая до сих пор поддерживается проектом GNU).

После RCS появилась CVS, Concurrent Versioning System, впервые вышедшая в 1986 году как набор скриптов для работы с файлами версий RCS в группах. Большим нововведением в CVS стало то, что в CVS несколько пользователей могут одновременно редактировать файл, а слияние изменений будет выполнено позже (одновременная правка). Появление такой возможности требовало обработки конфликтов редактирования. Разработчики могут отправлять в репозиторий новую версию какого-либо файла только, если она основана на последней доступной в репозитории версии этого файла. Если в репозитории и в моей рабочей копии есть различия, я должен устраниить все конфликты между файлами в них (т.е. от правок, затрагивающих те же самые строки).

CVS также привнесла идею *веток* (*branches*), которые позволяют разработчикам работать параллельно над различными частями кода, и тэгов, которые дают возможность согласованного обозначения версий, что позволяет легко на нее ссылаться. Хотя изначально изменения в CVS передавались через репозиторий, расположенный на файловой системе с общим доступом, в какой-то момент в CVS была реализована клиент-серверная архитектура для использования в больших сетях (таких как Интернет).

В 2000 году три разработчика собрались вместе, чтобы написать новую систему контроля версий, лишенную некоторых существенных недостатков CVS. Ее окрестили Subversion. Одним из главных отличий новой системы стало то, что Subversion работает с целыми деревьями за один раз, это означает, что изменения в версиях должны быть атомарными, последовательными, изолированными и долговременными. Рабочие копии Subversion также сохраняют первоначальные копии полученных из репозитория изменений, поэтому обычная операция *diff* (сравнение локального дерева с исходным) выполняется очень быстро.

Одной из интересных концепций в Subversion стало то, что тэги и ветки являются частью дерева проекта. Проект в Subversion обычно разделен на три части: *тэги*, *ветки* и *ствол* (*trunk*). Это решение оказалось интуитивно понятным для пользователей, которые не были знакомы с системами контроля версий, хотя гибкость присущая данному дизайну принесла немало проблем для инструментов конвертации, потому что тэги и ветки имеют более структурированное представление в других системах.

Все вышеназванные системы являются *централизованными*; начиная с CVS, все они знают как обмениваться изменениями между собой, при этом они используют какой-то определенный компьютер, на котором отслеживается история изменений репозитория. *Распределенная* система контроля версий вместо этого хранит копию всей (или большей части) истории из репозитория на каждом компьютере, у которого есть его рабочая копия.

12.1.2. Распределенный контроль версий

Несмотря на то, что Subversion явно превосходила CVS, у нее все равно был ряд недостатков.

Прежде всего во всех централизованных системах контроля версий фиксирование изменений (операция **commit**) и их публикация по сути являются одним и тем же, так как история репозитория хранится централизованно в одном месте. Это означает, что отправка изменений без доступа к сети невозможна.

Во-вторых, доступ к репозиториям в централизованных системах всегда требует передачи данных по сети на сервер, что делает такие системы относительно медленными по сравнению с локальным доступом в распределенных системах.

В-третьих, описанные ранее системы имели определенные недостатки при отслеживании слияний (хотя в некоторых из них с тех пор их поддержка улучшилась). Для больших групп разработчиков, работающих одновременно, важно, чтобы система контроля версий записывала, какие изменения были включены в новые версии кода, чтобы ничего не потерялось и последующие слияния могли использовать эту информацию.

В-четвертых, централизация, которая требуется традиционным системами контроля версий, кажется искусственной и выдвигает требование наличие единого пространства для интеграции. Сторонники распределенных системных контролей версий утверждают, что распределенные системы позволяют более органично организовать работу: разработчики могут передавать и интегрировать изменения так, как того требует проект в каждый момент времени.

Для решения описанных выше проблем появилось несколько инструментов. С моей позиции (позиции open-source разработчика) самыми главными на 2011 год стали Git, Mercurial и Bazaar. Проекты Git и Mercurial были начаты в 2005 году, когда разработчики ядра Linux решили больше не использовать проприетарную систему BitKeeper. Обе были начаты разработчиками Linux (Линусом Торвальдсом и Мэттом Маколлом, соответственно) с целью создания систем контроля версий, которые могли бы работать с сотнями тысяч изменений в десятках тысяч файлов (например, с ядром Linux). И на Мэтта, и на Линуса оказала влияние Monotone VCS. Bazaar разрабатывалась отдельно, но стала широко использоваться примерно в это же время, так как была использована компанией Canonical для всех своих проектов.

Создание распределенной системы контроля версий, естественно, представляет определенные сложности, многие из которых присущи всем распределенным системам. Для начала, в то время как в централизованных системах на сервере всегда находилась каноническая версия истории изменений, в распределенных системах такой версии нет. Изменения могут фиксироваться параллельно, что делает невозможным сортировку изменений по времени в каждом отдельно взятом репозитории.

Практически повсеместно для решения этой проблемы используется направленный ациклический граф изменений (DAG) вместо линейного упорядочения (Рисунок 1). То есть добавленное и зафиксированное изменение кода является потомком той версии кода, на основе которой оно было сделано, и никакая версия не может зависеть от самой себя или своих потомков. В этой схеме у нас есть три специальных типа версий кода: *корневая версия*, которая не имеет предков (репозиторий может иметь несколько корней), *объединяющая ревизия* (у которой несколько предков) и *главная версия*, у которой нет потомков. Каждое хранилище начинается с пустой корневой версии кода

и далее продолжается от нее по нескольким линиям изменений, заканчиваясь одной или несколькими главными версиями. Если два пользователя независимо друг от друга закоммитили свои изменения, и один из них хочет получить изменения кода от другого, то ему придется явно объединить изменения, внесенные другим пользователем, в новую версию, который он затем коммитит как объединяющую версию.

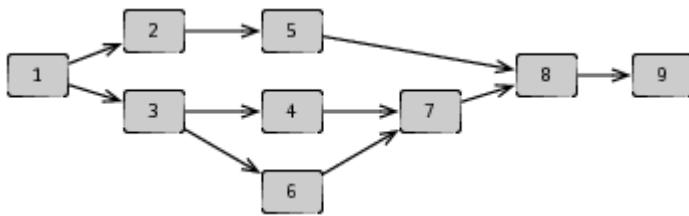


Рис.12.1: Направленный ациклический граф версий

Обратите внимание, что модель данного графа помогает решить некоторые проблемы, которые вызывают затруднения в централизованных системах: объединяющие версии используются для записи информации о вновь объединенных ветках графа. Получающийся в итоге граф может также полезно изображать большую группу параллельных веток, соединенных в меньшие группы, которые в конце концов будут соединены в одну специальную ветку, считающуюся канонической.

Этот подход требует, чтобы система отслеживала, что является предком каждого изменения в коде. Для облегчения обмена данными между изменениями, каждое из них обычно хранит информацию о своих предках, поэтому каждое изменение обычно имеет идентификатор. Хотя некоторые системы используют UUID или подобные схемы, Git и Mercurial предпочли SHA1-хэши содержимого. При этом дополнительным полезным свойством становится то, что ID может использоваться для верификации самого набора изменений. В действительности, так как информация о предках включается в захэшированные данные, вся история, идущая вплоть до любой версии, может быть проконтролирована при помощи ее хэша. Имена авторов, сообщения при коммите изменений, временные метки и другие метаданные хэшируются также, как и само содержимое файлов новой версии, поэтому они также могут быть верифицированы. А так как временные метки записываются во время фиксации, они также не обязательно идут друг за другом в каждом конкретном репозитории.

Все это может быть довольно сложным для людей, которые прежде пользовались только централизованными VCS: нет одного целого числа для обозначения версии, только 40-символьная шестнадцатиричная строка. Кроме того, больше нет глобальной сортировки, только локальная; вместо глобальной линейной сортировки есть только направленный упорядоченный график. Случайное создание нового направления разработки при отправке изменения в родительскую версию, у которой уже есть одно направление-потомок, может приводить в недоумение, если вы привыкли получать предупреждение от системы контроля версий, когда подобная ситуация происходила раньше.

К счастью, существуют инструменты для помощи в визуализации дерева, и Mercurial предоставляет возможность использования коротких версий хэша, а также чисел для идентификации локальных изменений. В последнем случае используется увеличивающееся целое число, которое отображает порядок, в котором изменения были добавлены в копию. Так как данный порядок может быть различным от копии к копии, его нельзя использовать в нелокальных операциях.

12.2. Структуры данных

Теперь, когда концепция направленного ациклического графа должна быть достаточно ясной, давайте рассмотрим как эти графы хранятся в Mercurial. Модель графа является центральной во внутреннем механизме Mercurial, в действительности мы используем несколько различных графов в хранилище репозитория на диске (а также в структуре кода в памяти). В этой секции объясняется, что это за структуры и как они взаимодействуют вместе.

12.2.1. Проблемы

До того, как мы окунемся в информацию о структурах данных, я бы хотел немного рассказать об окружении, в котором развивался Mercurial. Первое упоминание о Mercurial может быть найдено в сообщении, которое Мэтт Маколл отоспал в список рассылки ядра Linux в апреле 2005 года. Это произошло вскоре после того, как было решено, что BitKeeper больше не может быть использован для развития ядра. Мэтт начал свое сообщение, обозначив некоторые цели: система должна быть простой, масштабируемой и эффективной.

В своем докладе «К лучшему управлению источниками данных: Revlog и Mercurial» в 2006 году Мэтт заявил, что современная система контроля версий должна работать с деревьями, состоящими из миллиона файлов, работать с миллионами изменений и масштабироваться между тысячами пользователей, создающими новые версии кода параллельно в течение десятилетий. Исходя из этих целей он указал лимитирующие технологические факторы

- скорость: ЦПУ
- объем: дисковое пространство и память
- пропускная способность: память, LAN, диск и WAN
- скорость чтения диска

Скорость чтения диска и пропускная способность WAN являются сегодня ограничивающими факторами, и поэтому система должна быть оптимизирована с их учетом. В докладе также излагаются обычные сценарии или критерии для оценки производительности таких систем на уровне файлов:

- Компрессия хранилища: какая форма компрессии больше всего подходит для хранения истории файлов на диске? Какой алгоритм позволяет получить наилучшую производительность ввода/вывода, при этом не допуская того, чтобы процессор стал узким местом системы?
- Получение произвольных версий файлов: некоторые системы контроля версий хранят конкретную версию таким образом, что для того, чтобы воссоздать новую версию (с использованием информации об изменениях), будет необходимо прочитать большое количество более старых версий кода. Необходимо контролировать этот процесс, чтобы получение старых версий оставалось быстрым.
- Добавление версий файлов: мы регулярно добавляем новые версии. Мы не хотим перезаписывать старые версии каждый раз, когда мы добавляем новую, потому что это существенно замедлит процесс, когда версий станет достаточно много.
- Показ истории файлов: мы хотим иметь возможность просматривать историю всех изменений, которые происходили с конкретным файлом. Это также позволит делать аннотации (которые назывались *blame* в CVS, но были переименованы в *annotate* в более поздних системах, чтобы убрать негативный подтекст): просмотреть исходный вид каждой строки в файле до изменения.

В докладе также рассматриваются подобные сценарии на уровне проекта. Основными операциями на этот уровень являются получение версии, коммит изменений (создание новой версии), поиск изменений в рабочей копии. Последняя операция, в частности, может быть медленной для больших деревьев (для проектов типа Mozilla или NetBeans, каждый из которых использует Mercurial для контроля версий).

12.2.2. Быстрое хранение версий: Revlogs

Решение, которое нашел Мэтт для Mercurial, было названо *revlog* (сокращенно от *revision log* – лог версий). Revlog — это способ эффективного хранения версий файлов (каждая из версий при этом содержит определенные изменения по сравнению с предыдущей). Такой способ должен быть эффективным с точки зрения времени доступа (то есть оптимизированным для поиска по диску) и занимаемого дискового пространства, учитывая обычные сценарии, описанные в предыдущей

секции. Чтобы удовлетворить этим критериям, revlog представляет собой два файла на диске: индекс и файл данных.

Таблица 1: Формат записи Mercurial

6 байт	Смещение фрагмента
2 байта	Флаги
4 байта	Длина фрагмента
4 байта	Длина в несжатом виде
4 байта	Базовая версия
4 байта	Связанная версия
4 байта	Версия-предок 1
4 байта	Версия-предок 2
32 байта	Хэш

Индекс состоит из записей фиксированной длины, содержимое которых описано в таблице 1. То, что записи имеют фиксированную длину, позволяет прямое (т.е. за постоянное время) обращение к версии по ее локальному номеру: мы можем просто прочитать файл индекса до определенной позиции (которая вычисляется как «длина записи в индексе умноженная на номер версии») для нахождения нужных данных. Разделение индекса и данных также означает, что мы можем быстро прочитать данные индекса без необходимости поиска по диску по всему файлу данных.

Смещение фрагмента и длина фрагмента указывают на фрагмент файла данных, который нужно прочитать, чтобы получить сжатые данные для данной версии. Интересно то, как определяется, когда нужно сохранить новую базовую версию. Это решение основывается на сравнении совокупного размера изменений и размера несжатой версии (данные сжимаются при помощи zlib, чтобы занимать на диске еще меньше места). Ограничивая таким образом размер цепочки изменений, мы точно знаем, что восстановление данных для конкретной версии не потребует чтения слишком большого количества информации об изменениях.

Поле «связанные версии» используется для того, чтобы зависящий revlog указывал на revlog более высокого уровня (подробнее об этом будет рассказано далее), родительские версии хранятся с указанием локального целочисленного номера версии. Опять же это делает простым поиск соответствующих данных через revlog. Хэш используется для хранения уникального идентификатора конкретного внесенного изменения. Мы используем 32 байта вместо 20 байт, требуемых для SHA1, для поддержки возможных изменений в будущем.

12.2.3. Три revlog

Основная структура данных истории хранится в revlog, на ее основе мы можем создать модель данных для нашей файловой структуры. Модель состоит из трех типов revlog: *лог изменений*, *манифесты* и *файловый лог*. Лог изменений содержит метаданные для каждой версии с указателем на манифест (то есть с идентификатором узла одной версии в манифесте). В свою очередь манифест представляет собой список имен файлов, каждому из которых сопоставлен идентификатор узла, указывающий на версию в файловом логе. В коде у нас созданы классы для лога изменений, манифеста и файлового лога, каждый из которых является потомком общего класса revlog, что понятно отражает обе концепции.

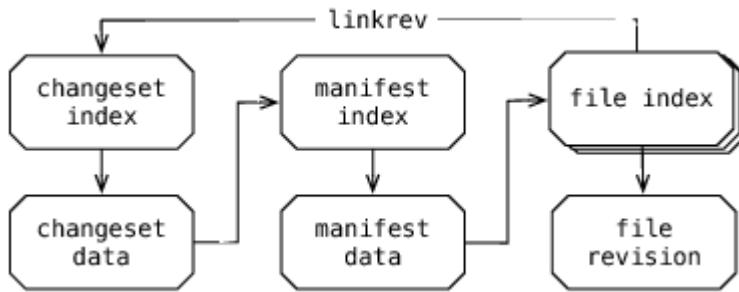


Рис.12.2: Структура лога

Версия лога изменений выглядит следующим образом:

```

0a773e3480fe58d62dcc67bd9f7380d6403e26fa
Dirkjan Ochtman
1276097267 -7200
mercurial/discovery.py
discovery: fix description line
  
```

Эти данные вы получаете из слоя revlog; слой лога изменений превращает его в простой список значений. Первая строка содержит хэш манифеста, затем идет имя автора, дата и время (в форме метки времени Unix и смещения временной зоны), список затронутых файлов и сообщение с описанием. Здесь скрыта одна вещь: мы разрешаем добавление произвольных метаданных в лог изменений, и для того, чтобы сохранить обратную совместимость, эти данные идут после метки времени. Затем идет манифест:

```

.hgignore\x006d2dc16e96ab48b2fccaa44f7e9f4b8c3289cb701
.hgsigs\x00de81f258b33189c609d299fd605e6c72182d7359
.hgtags\x00b174a4a4813ddd89c1d2f88878e05acc58263efa
CONTRIBUTORS\x007c8afb9501740a450c549b4b1f002c803c45193a
COPYING\x005ac863e17c7035f1d11828d848fb2ca450d89794
...
  
```

Это версия манифеста, на которую указывает изменение 0a773e (интерфейс пользователя Mercurial позволяет сокращать идентификатор до любого префикса, по которому можно однозначно идентифицировать данные). Это простой список всех файлов в дереве, один файл в каждой строке, где после имени файла идет нулевой байт, затем шестнадцатиричный идентификатор узла, который указывает на файловый лог данного файла. Директории в дереве представлены не отдельно, а просто подразумеваются за счет слэшей в путях. Помните, что изменения в манифесте хранятся в хранилище аналогично любому другому revlog, поэтому данная структура дает возможность легко слою revlog хранить только измененные файлы и их новые хэши для любой новой версии. Манифест обычно представлен структурой данных типа «таблица хэшированных значений» в коде Python, при этом имена файлов являются ключами, а узлы — значениями.

Третий тип revlog — это файловый лог. Файловые логи хранятся во внутренней папке Mercurial под названием *store*, при этом они называются практически также, как и файлы, которые они отслеживают. Имена немного кодируются, чтобы сохранить кроссплатформенность между основными операционными системами. Например, нам приходится иметь дело с зависимостью от регистра символов в названии файлов и папок на Windows и Mac OS X, неразрешенными именами файлов в Windows и различными кодировками, используемыми в разных файловых системах. Как вы можете себе представить, сделать надежной кроссплатформенную работу довольно сложно. С другой стороны содержимое версии файлового лога не настолько интересно: просто содержимое файла плюс некоторые дополнительные префиксы метаданных (которые мы используем для отслеживания копий файлов и файлов с тем же именем, помимо прочего).

Эта модель дает нам полный доступ к хранилищу данных в репозитории Mercurial, но она не всегда является очень удобной. В то время как реальная используемая модель ориентирована верти-

кально (один файловый лог на файл), разработчики Mercurial часто понимают, что им хотелось бы работать со всеми деталями из одной версии кода: они начинают с изменений из лога изменений и им нужен простой доступ к манифесту и файловому логу из той же версии. Позже был добавлен новый набор классов, по иерархии располагающийся поверх revlog. Так появились *контексты*.

Одним из преимуществ существующего способа создания отдельных revlog является очередность их создания. Сначала добавляется информация в файловый лог, затем в манифест, наконец, в лог изменений, поэтому данные в репозитории всегда в целостном состоянии. Любой процесс, начинаящий читать лог изменений, может быть уверен в том, что все указатели на другие revlog являются, это снимает множество проблем в этой области. Однако, в Mercurial есть несколько явных блокировок для предотвращения параллельного добавления данных в revlog двумя процессами.

12.2.4. Рабочая копия

Последняя важная структура данных называется *dirstate*. Dirstate по сути является представлением того, что находится в рабочей папке в каждый момент времени. Более того, в ней отслеживаются изменения того, какая версия кода была отправлена: это отправная точка для всех сравнений при помощи команды *status* или *diff*, кроме того она определяет родительскую версию (или версии) для следующих изменений, которые будут закоммичены. В dirstate будет создан набор из двух родителей каждый раз, когда будет применена команда *merge* с целью объединить одни изменения с другими.

Так как команды *status* и *diff* являются очень распространенными (они помогают отслеживать прогресс в том, что у вас есть сейчас по сравнению с предыдущими изменениями), *dirstate* также содержит кэш состояния рабочей копии с последнего раза, когда она была прочитана Mercurial. Отслеживание времени последней модификации и размеров файлов позволяет ускорить обход структуры файлов. Нам также необходимо отслеживать состояние файла: был ли он добавлен, удален или объединен в рабочей директории. Это также позволит ускорить обход рабочей копии и упростит получение данной информации в момент коммита изменения.

12.3. Механизм контроля версий

Теперь, когда вы знакомы с лежащими в основе моделями данных и структурой кода на низком уровне, давайте перейдем на более высокий уровень и рассмотрим, как в Mercurial реализованы концепции контроля версий.

12.3.1. Ветки

Ветки используются повсеместно для разделения различных направлений разработки, которые позже будут интегрированы. Это может быть необходимо, если кто-то экспериментирует с новым подходом, чтобы у нас всегда оставалась главное направление разработки в доступном виде (“ветки для новых функций”), или для того, чтобы быстро выпускать релизы с исправлениями для старых версий (“поддерживающие ветки”). Оба подхода часто используются и поддерживаются всеми современными системами контроля версий. В то время как неявные ветки являются распространенными в системах на основе направленного ациклического графа, именованные ветки (такие, где имя ветки хранится в метаданных изменений) не столь распространены.

Изначально, в Mercurial не было способа явно указывать имена веток. Вместо этого ветки создавались путем создания новых копий и их раздельной публикации. Такой путь эффективен и легок для понимания, он особенно полезен для веток нового функционала, потому что накладные расходы на эти операции не велики. Однако в больших проектах создание копий может быть довольно затратным: так как хранилище репозитория привязано к файловой системе, создание отдельной рабочей копии является медленным процессом и может потребовать большое дисковое пространство.

Из-за этих недостатков, в Mercurial был добавлен второй способ создания ветвей: добавление имени ветки в метаданные изменения. Появилась команда `branch`, с ее помощью вы можете задать имя ветки для текущей рабочей директории, тогда это имя будет использовано для коммита следующего изменения. Обычная команда `update` может быть использована для обновления имени ветки; изменение, закоммиченное в ветке, всегда будет привязано к конкретной ветке. Этот подход называется *именованные ветки*. Однако, прошло несколько релизов Mercurial прежде чем появилась поддержка закрытия таких веток (закрытие скрывает ее из списка веток). Закрытие реализовано путем добавления дополнительного поля в метаданные изменения, в котором записывается, что данное изменение закрывает ветку. Если у ветки более, чем одно направление, все они должны быть закрыты перед тем, как ветка исчезнет из списка веток репозитория.

Конечно, существует более чем один способ реализации веток. В Git используется другой способ наименования веток: при помощи ссылок. Ссылки — это имена, указывающие на другой объект в истории Git, обычно на изменение. Это означает, что ветки в Git эфемерные: как только вы убираете ссылку, информация о том, что ветка когда-то существовала, исчезает без следа; это похоже на то, что вы получаете, когда используете отдельную копию Mercurial и объединяете ее обратно с другой точной копией. Это позволяет очень легко и без затрат манипулировать ветками локально, и предотвращает перегруженность списка веток.

Этот способ создания веток оказался очень популярным, гораздо более популярным, чем именованные ветки или клоны веток в Mercurial. Это привело к созданию расширения `bookmarksq`, которое, вероятно, будет добавлено в Mercurial в будущем. Оно использует простой неверсионный файл для отслеживания ссылок. Сетевой протокол, используемый для обмена данными в Mercurial, был расширен для добавления возможности передачи информации о ссылках.

12.3.2. Тэги

На первый взгляд то, как в Mercurial реализованы тэги, может быть несколько странным. Когда вы в первый раз добавляете тэг (используя команду `tag`) в репозиторий будет добавлен и закоммичен файл `.hgtags`. Каждая строка в этом файле содержит идентификатор узла изменения и имя тэга для него. Таким образом, файл тэгов обрабатывается также, как и любой другой файл в репозитории.

На это есть три важные причины. Первая состоит в том, что необходимо иметь возможность изменять тэги; случаются ошибки и нужно, чтобы их можно было исправить или удалить. Вторая, тэги должны быть частью истории изменений: важно видеть, когда был создан тэг, кем и по какой причине, или даже когда тэг был изменен. Третья причина: нужно иметь возможность добавить тэг к изменению, имевшем место в прошлом. Например, некоторые проекты активно тестируют эталон релиза из системы контроля версий до того, как выпустить его.

Все эти свойства легко возможны при наличии данной архитектуры с файлами `.hgtags`. Хотя некоторые люди не понимают наличия файлов `.hgtags` в своих рабочих копиях, такие файлы делают интеграцию механизма тэгирования с другими частями Mercurial (например, синхронизацию с другой копией репозитория) очень простой. Если бы тэги существовали вне дерева исходных кодов (как в Git, к примеру), был бы необходим отдельный механизм для контроля источников тэгов и решения конфликтов от дублированных тэгов. Даже несмотря на то, что последняя ситуация не является очень частой, хорошо иметь такую архитектуру, когда подобные проблемы даже не возникают.

Чтобы все это работало правильно, Mercurial только один раз добавляет новые строки в файл `.hgtags`. Это также облегчает операцию слияния, если тэги были созданы параллельно в разных копиях. Более новый идентификатор узла для каждого конкретного тэга всегда имеет преимущество, а добавление нулевого идентификатора узла (представляющего пустую корневую версию, которая является общей для всех репозиториев) приведет к удалению тэга. Mercurial также принимает во внимание тэги из всех веток в репозитории, используя анализ, какие из них более новые, для определения их порядка.

12.4. Общая структура

Mercurial почти полностью написан на Python, только некоторые части написаны на C (там, где критична производительность всего приложения). Python оказался наиболее подходящим для большей части кода, потому что гораздо легче выражать высокоуровневые концепции на динамическом языке, вроде Python. Так как для большей части кода быстродействие не слишком критично, мы не против того, что нас будут критиковать за то, что мы кое-где упростили себя процесс разработки в ущерб производительности.

Модуль в Python находится в одном файле кода. Модули могут содержать столько кода, сколько необходимо, и поэтому являются важным способом организации программы. Модули могут использовать типы или вызывать функции из других модулей путем явного импорта других модулей. Директория, содержащая файл `__init__.py`, является пакетом, и все модули пакета будут доступны интерпретатору Python.

По умолчанию Mercurial устанавливает два пакета в окружение Python: **mercurial** и **hgext**. Пакет **mercurial** содержит ядро, необходимое для запуска Mercurial, в то время как **hgext** содержит некоторые расширения, которые считаются достаточно полезными и поэтому распространяются вместе с ядром. Однако, если они нужны, их все же нужно вручную включить в конфигурационном файле (это будет описано позже).

Для простоты Mercurial является консольным приложением. Это означает, что у него простой интерфейс: пользователь вызывает скрипт hg и команду. Эта команда (например, log, diff или commit) может принимать несколько опций или аргументов; некоторые опции подходят для всех команд. В том, что касается интерфейса, есть три нюанса:

- **hg** часто выводит информацию, которую запросил пользователь, или показывает сообщения о статусе
- **hg** может запросить дальнейшую информацию через консоль
- **hg** может запускать внешнюю программу (редактор для создания сообщения о коммите изменения или программу, для улаживания конфликтов в коде при слиянии)

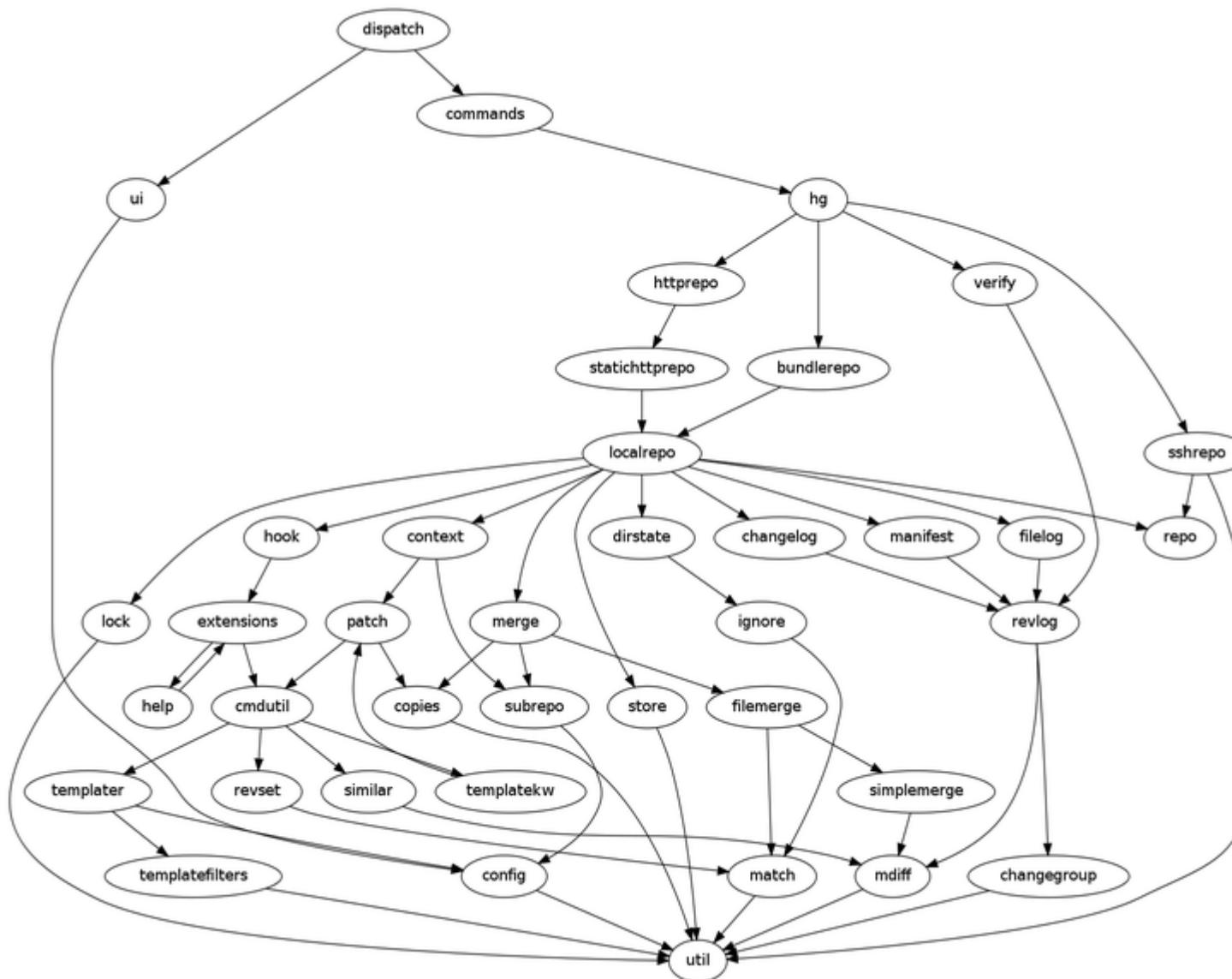


Рис.12.3: Граф импорта

Начало этого процесса можно удобно наблюдать с помощью графа импорта на рисунке 3. Все аргументы командной строки передаются в функцию в модуль-диспетчер (**dispatch**). Первое что происходит: создается экземпляр объекта **ui**. Класс **ui** попытается найти конфигурационные файлы в нескольких хорошо известных местах (таких, как ваша домашняя папка) и сохранить параметры конфигурации в объекте **ui**. Конфигурационные файлы могут содержать пути к расширениям, которые также должны быть загружены в этот момент. Любой глобальный параметр, переданный через командную строку, также сохраняется в объекте **ui** на этом этапе.

После того, как все это будет выполнено, нам нужно решить, нужно ли создавать объект репозитория. В то время, как большинство команд требуют наличия локального репозитория (представлен классом **localrepo** из модуля **localrepo**), некоторые команды могут работать с удаленными репозиториями (через HTTP, SSH или каким-либо другим зарегистрированным образом), а некоторые команды могут выполнять свою работу вообще без обращения к репозиторию. Последняя категория включает команду **init**, которая используется для инициализации нового репозитория.

Все ключевые команды ядра представлены одной функцией в модуле **commands**; это дает возможность очень легко находить код каждой функции. Модуль команд также содержит хэшированную таблицу, которая хранит соответствие между именем команды, функцией и принимаемыми ей параметрами. То, как это выполнено, дает возможность наличия общего набора параметров у разных опций (например, у многих команд параметры аналогичны команде **log**). Описания параметров позволяют модулю-диспетчеру проверять конкретную опцию для любой из команд и кон-

вертировать любые значения в тип, ожидаемый функцией команды. Практически каждая функция также получает доступ к объекту `ui` и объекту `repository`.

12.5. Расширяемость

Одним из факторов, который делает Mercurial очень мощным, является возможность написания расширений под него. Так как Python является языком, начать писать на котором довольно просто, а API Mercurial по большей части хорошо спроектирован (хотя и не всегда полностью документирован), некоторые люди начали писать на Python в первый раз именно потому, что они захотели написать расширение для Mercurial.

12.5.1. Написание расширений

Расширения включаются путем добавления строки в один из конфигурационных файлов, которые Mercurial читает при загрузке. Есть несколько способов добавить функционал:

- добавление новой команды;
- создание оболочки для существующей команды;
- создание оболочки для используемого репозитория;
- обертка любой функции в Mercurial;
- добавление новых типов репозитория.

Добавление новых команд может быть выполнено просто путем добавления хэшированной таблицы под названием `cmdtable` в модуль расширения. Он будет подгружен загрузчиком расширений, который добавит новую команду в таблицу команд, используемую при обработке команд. Аналогично, расширения могут определять функции `uisetup` и `reposetup`, которые вызываются в коде диспетчера, после того как созданы объекты пользовательского интерфейса и репозитория. Типичным способом является использования функции `reposetup` для обравчивания репозитория в подкласс репозитория, предоставляемый расширением. Это позволяет расширению модифицировать базовое поведение. Например, одно расширение, которое я написал, подцепляет `uisetup` и устанавливает свойство конфигурации `ui.username` в зависимости от данных доступа SSH, доступных из окружения.

Более серьезные расширения могут быть написаны для добавления типов репозиториев. Например, проект `hgsubversion` (не включенный в Mercurial) регистрирует тип репозитория Subversion. Это делает возможным клонирование репозитория Subversion, как будто это репозиторий Mercurial. Существует даже возможность обратной отправки данных в репозиторий Subversion, хотя в определенных случаях возможны проблемы из-за различий в двух системах. Пользовательский интерфейс, с другой стороны, полностью прозрачен.

Для тех кто хочет глубоко изменить Mercurial, в мире динамических языков существует такая вещь как «monkeypatching». Так как код расширений выполняется в том же адресном пространстве, что им Mercurial, а Python — это достаточно гибкий язык с широкими рефлексивными возможностями, возможно (и достаточно легко) модифицировать любой класс или функцию, определенную в Mercurial. Хотя это может привести к довольно уродливым хакам, это действительно мощный механизм. Например, расширение `highlight` (представлено в `hgext`) модифицирует встроенный веб-сервер, чтобы добавить подсветку синтаксиса на страницы браузера репозитория, что позволяет вам просматривать содержимое файлов.

Есть еще один способ расширения функциональности Mercurial, гораздо более простой: *алиасы*. Любой конфигурационный файл может определять алиас в качестве нового имени для существующей команды с определенным набором заданных опций. Это также дает возможность давать более короткие названия любой команде. Последние версии Mercurial также включают возмож-

ность вызова команд шелла через алиасы, так что вы можете создавать сложные команды не используя ничего, кроме шелл-скриптов.

12.5.2. Перехватчики

Системы контроля версий давно предоставляли механизм перехватчиков в качестве способа взаимодействия событий этих систем с окружающим миром. Обычной практикой является отправка уведомляющего сообщения в систему постоянной интеграции или обновление рабочей копии на веб-сервере таким образом, чтобы изменения стали видимы во всем мире. Конечно, такие возможности присутствуют и у Mercurial.

В действительности, здесь также присутствуют два варианта. Один больше похож на традиционные перехватчики в других системах контроля версий, он вызывает скрипты в шелле. Другой более интересен, потому что позволяет пользователям вызывать Python-овские перехватчики, указывая модуль Python и имя функции из этого модуля для вызова. Это не только быстрее, потому что работает в том же процессе, но при этом также используются объекты `repo` и `ui`, то есть вы можете легко создавать более сложное взаимодействие внутри системы контроля версий.

Перехватчики в Mercurial могут быть разделены на предкомандные, посткомандные, управляющие и дополнительные. Первые две категории просто определяются путем указания ключей `pre-command` или `post-command` в секции перехватчиков в конфигурационном файле. Для двух других типов есть предопределенный набор событий. Различие в управляющих перехватчиках состоит в том, что они запускаются прямо перед тем, как что-то происходит, и могут не позволить этому событию выполняться дальше. Они обычно используются для валидации изменений каким-либо образом на центральном сервере; по причине распределенной природы Mercurial такие проверки не могут быть выполнены во время коммита изменений. Например, проект Python использует перехватчик, для проверки применения некоторых аспектов стиля кода — если изменение добавляет код, который имеет не разрешенный стиль, он будет отвергнут центральным репозиторием.

Еще один интересный способ использования перехватчиков — это `pushlog`, который используется в Mozilla и некоторых других компаниях. Pushlog записывает каждое добавление кода (так как при этом может содержаться любое число изменений) и записывает, кто инициировал это добавление и когда, то есть создается своего рода отчетность по использованию репозитория.

12.6. Выводы

Одним из первых решений, которое принял Мэтт, когда начинал разработку Mercurial, было то, что разрабатывать надо на Python. Python зарекомендовал себя с самой лучше стороны с точки зрения расширяемости (через расширения и перехватчики), и на нем очень легко писать код. Он также берет на себя большую часть работы по совместимости кода между различными платформами, что достаточно легко дает возможность Mercurial успешно работать с тремя главными операционными системами. С другой стороны Python медленный, по сравнению со многими другими (компилирующимися) языками; в частности запуск интерпретатора достаточно медленный, что не очень хорошо для инструментов, которые запускаются много и часто (таких как система контроля версий), а не работают как долговременные процессы.

Также одним из ранних решений была сложность модификации изменений после их коммита. Так как невозможно изменить версию без модификации ее идентифицирующего хэша, «обратный вызов» изменений после их публикации в Интернете приносит проблемы, и поэтому Mercurial не дает выполнить такую операцию. Однако, изменение незафиксированных версий не должно быть затруднено, и сообщество пыталось упростить этот процесс вскоре после релиза. Существуют расширения, которые пытаются решить эту проблему, но они требуют определенного обучения и не являются интуитивно понятными для пользователей, которые до этого пользовались только основными функциями Mercurial.

Revlog хороши тем, что позволяют уменьшить количество обращений к диску; послойная архитектура лога изменений, манифеста и файлового лога также показала себя с хорошей стороны. Коммит изменений осуществляется быстро и достаточно мало дискового пространства используется для хранения версий. Однако, некоторые операции, например, переименование файлов, выполняются не очень эффективно из-за раздельного хранения версий для каждого файла; в конце концов это будет исправлено, но потребует какого-то нестандартного хака, нарушающего текущую концепцию слоев. Аналогично, пофайловый направленный ациклический граф, используемый для помощи в хранилища файлового лога, не используется на практике достаточно широко, поэтому код, используемый для работы с этими данными может быть посчитан излишним.

Еще одним ключевым фактором для Mercurial была необходимость легкости в обучении. Мы старались собрать большинство требуемых функций в небольшой набор команд со схожими опциями. Идея состояла в том, чтобы Mercurial можно было изучить прогрессивно, особенно тем пользователям, которые использовали другие системы контроля версий до этого; эта философия расширяется до такой степени, что расширения могут быть использованы для модификации Mercurial еще больше в каждом конкретном случае. По этой причине разработчики также пытались создать пользовательский интерфейс, схожий с другими системами контроля версий, Subversion в частности. Также команда попыталась предоставить хорошую документацию, доступную из самого приложения, со ссылками на другие части справки и информацию о командах. Мы следим за тем, чтобы сообщения об ошибках имели смысл, включая подсказки что еще можно сделать вместо той операции, которая завершилась ошибкой.

Некоторые менее важные решения могут быть удивительными для новых пользователей. Например, управление тэгами (как обсуждалось в предыдущей секции) путем хранения их в отдельном файле внутри рабочей директории не понравилось по началу многим пользователям, но данный механизм имеет некоторые весьма ощутимые преимущества (хотя и не лишен недостатков). Кроме того, другие системы контроля версий по умолчанию посыпают на сервер изначально загруженные изменения и их предков, в то время как Mercurial посыпает каждое зафиксированное изменение, которого нет на удаленном сервере. Оба подхода имеют определенный смысл, и каждый подходит для определенного стиля разработки.

Как и в любом программном проекте, нам пришлось идти на множество компромиссов. Я считаю, что в случае Mercurial мы во многом делали правильный выбор, хотя если оглянуться назад, безусловно, что-то можно было сделать иначе. Исторически, Mercurial оказался частью первого поколения распределенных систем контроля версий, достаточно зрелых для широкого использования. Что касается меня, то я хотел бы увидеть, как будет выглядеть следующее поколение таких систем.

13. Экосистема NoSQL

Глава 13 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

В отличие от большинства других проектов, описанных в данной книге, NoSQL является не инструментом, а экосистемой, сформированной несколькими аналогичными и конкурирующими друг с другом инструментами. Программные компоненты из категории NoSQL являются альтернативой реляционным системам баз данных, используемых для хранения данных и использующих язык SQL для запросов. Для понимания принципов работы систем NoSQL нам придется понять принцип работы ряда доступных систем и рассмотреть вопрос о том, как архитектура каждой из этих систем расширяет набор ее возможностей для хранения данных.

Если вы обдумываете возможность использования системы NoSQL для хранения данных, вам в первую очередь придется разобраться в обширном наборе систем NoSQL. Системы NoSQL лишены множества полезных возможностей, присущих традиционным базам данных, и операции, обычно реализуемые в рамках системы базы данных, в данном случае возлагаются на разработчи-

ков приложений. Это потребует от вас дополнительной работы в роли системного архитектора, что в свою очередь предусматривает более глубокое понимание принципов построения таких систем.

13.1. Что в имени?

Перед рассмотрением систем NoSQL, давайте сначала разберемся с их названием. Образно говоря, система NoSQL предоставляет пользователю интерфейс запросов, не использующий язык SQL. Сообщество разработчиков систем NoSQL в общем случае рассматривает их более глобально и заявляет, что системы NoSQL являются альтернативой традиционным реляционным базам данных и позволяют разработчикам проектировать приложения, использующие *не только* интерфейс SQL-запросов. В некоторых случаях вы сможете заменить реляционную базу данных на альтернативную систему NoSQL, а в некоторых вам придется применять метод комбинирования и подбора систем (*mix-and-match approach*) для решения различных проблем, с которыми вы столкнетесь во время разработки приложения.

Перед погружением в мир систем NoSQL, давайте рассмотрим случаи, в которых язык SQL и реляционная модель будут соответствовать вашим требованиям, а также другие случаи, в которых система NoSQL может оказаться более предпочтительной.

13.1.1. Язык SQL и реляционная модель

SQL является декларативным языком для осуществления запросов данных. В декларативном языке разработчик задает действия, которые должна выполнить система вместо процедурного описания того, как система должна выполнить эти действия. Несколько примеров: поиск записи работника с идентификатором 39, выделение только информации о имени и номере телефона работника из его записи, вывод записей о работниках бухгалтерии, подсчет количества работников в каждом отделе или объединение данных из таблицы с информацией о работниках с данными из таблицы с информацией об управляющих.

В первом приближении язык SQL позволяет вам выполнить эти запросы без понимания того, как данные размещаются на диске, какие индексы используются для доступа к данным или какие алгоритмы используются для обработки данных. Важным архитектурным компонентом большинства реляционных баз данных является *оптимизатор запросов*, который принимает решение о том, какие из множества логически эквивалентных схем запросов следует выполнить для получения наиболее быстрого ответа на запрос. Эти оптимизаторы обычно работают лучше среднестатистического пользователя базы данных, но иногда они не располагают достаточным количеством информации или используют в значительной степени упрощенную модель системы, что затрудняет генерацию наиболее эффективного запроса.

Реляционные базы данных, наиболее часто используемые на практике, следуют *реляционной модели данных*. В рамках данной модели различные данные из реального мира хранятся в различных таблицах. Например, все данные работников могут храниться в таблице "Employees", а все данные отделов могут храниться в таблице "Departments". Каждая строка таблицы содержит различные свойства, хранимые в столбцах. Например, работники могут характеризоваться идентификаторами, жалованием, датами рождения, а также именами и фамилиями. Каждое из этих свойств будет храниться в столбце таблицы работников "Employees".

Реляционная модель идет рука об руку с языком запросов SQL. Простые запросы SQL, такие, как фильтры, извлекают все записи, поля которых соответствуют какому-либо выражению (т.е. идентификатор работника = 3 или жалование > \$20000). Более сложные запросы заставляют базу данных выполнить дополнительную работу, такую, как объединение данных из нескольких таблиц (т.е. как называется отдел, в котором числится работник с идентификатором 3?). Другие сложные

запросы, такие, как запросы вычисления сводных показателей (т.е. какова средняя зарплата моих работников?) могут приводить к полному обходу таблиц.

Реляционная модель данных описывает структурированные объекты с жесткими связями между ними. Запросы к этой модели посредством SQL позволяют осуществлять сложные манипуляции с данными без необходимости разработки сложных алгоритмов. Сложность таких моделей и запросов имеет ограничение, однако:

- Сложность ведет к непредсказуемости. Выразительность языка SQL затрудняет оценку нагрузки на систему в ходе выполнения каждого из запросов, а таким образом и оценку общей нагрузки. В то время, как более простые языки могут приводить к усложнению логики приложений, они упрощают работу систем хранения данных, которые отвечают только на простые запросы.
- Существует множество путей моделирования проблемы. Реляционная модель данных является строгой: схема, ассоциированная с каждой таблицей, задает данные в каждой строке. Если мы храним менее структурированные данные или строки со значительно различающимися столбцами, реляционная модель может оказаться излишне строгой. Аналогично, разработчики приложений могут не признать реляционную модель в качестве превосходного метода структурирования любых типов данных. Например, большая часть логики приложений разработана с использованием объектно-ориентированных языков программирования и использует такие высокуюровневые объекты, как списки, очереди и множества, а также некоторые разработчики могут пожелать продолжить использовать их постоянный уровень абстракции для ее моделирования.
- Если объем данных превышает доступный объем хранилища на сервере, таблицы из базы данных должны быть распределены между серверами. Для предотвращения объединений данных из таблиц, в ходе которых будет осуществляться их передача по сети с целью распределения данных по различным таблицам нам придется произвести денормализацию. Денормализация позволяет разместить данные из нескольких таблиц, которые могут потребоваться одновременно, в одном и том же месте. Это делает нашу базу данных похожей на систему хранения данных с идентификаторами в виде ключей, а нам остается только задуматься о том, какая из других моделей данных может подойти лучшим образом.

Отказ от рассмотрения результатов многолетнего проектирования без видимых на то причин не является разумным решением. Когда вы решаете хранить ваши данные в базе данных, рассматривайте возможность использования языка запросов SQL и реляционной модели, которые стали результатом многих лет исследования и разработки и предоставляют богатые возможности моделирования и понятные гарантии в отношении сложных операций. Системы NoSQL являются хорошим решением в том случае, если ваша задача является такой специфичной, как хранение больших объемов данных, работа под большими нагрузками или работа со сложной моделью данных, для чего язык SQL и реляционные базы данных могут быть не в достаточной степени оптимизированы.

13.1.2. Исходные данные для проектирования систем NoSQL

Системы NoSQL в большей степени создавались под влиянием документации от исследовательского сообщества. В то время, как многие документы описывали решения, положенные в основу архитектуры систем NoSQL, две системы следует выделить особо:

Система BigTable компании Google [[CDG+06](#)] представляет интересную модель данных, которая упрощает реализацию многоколоночного отсортированного хранилища данных истории. Данные распределяются по множеству серверов с использованием иерархической схемы распределения на основе диапазонов, а данные обновляются в строгой согласованности (подход, который мы обсудим позднее в Разделе 13.5).

Система Dynamo компании Amazon [[DHJ+07](#)] использует отличное от предыдущего распределенное хранилище данных с доступом на основе ключей. Модель данных системы Dynamo проще за счет установления соответствия ключей специфичным для приложений бинарным данным. Модель распределения данных более устойчива к ошибкам, но эта цель достигается путем применения подхода, заключающегося в снижении степени согласованности данных и называемого конечной согласованностью (eventual consistency).

Мы подробно рассмотрим каждое из этих решений, но важно понимать, что многие из них могут комбинироваться и подбираться друг для друга. Некоторые системы NoSQL, такие, как HBase^{[1](#)}, реализуют архитектуру, аналогичную BigTable. Другая система NoSQL под названием Voldemort^{[2](#)}

копирует многие возможности системы Dynamo. Также другие проекты NoSQL, такие, как Cassandra³, переняли ряд возможностей из системы BigTable (ее модель данных), а также ряд других возможностей из системы Dynamo (ее схемы распределения и поддержания согласованности данных).

13.1.3. Характеристики и соображения

Системы NoSQL оставляют в стороне мощный стандарт запросов SQL и являются более простым, но менее обобщенным решением для проектирования систем хранения данных. Эти системы были созданы с надеждой на то, что упрощение механизма обработки данных позволит архитектору лучше предугадать производительность каждого из запросов. В множестве систем NoSQL сложная логика запросов должна быть реализована на стороне приложения, в результате чего производительность запросов к хранилищу данных может быть оценена более точно из-за отсутствия значительных изменений в этих запросах.

Системы NoSQL реализуют больший набор функций, чем поддержка декларативных запросов для манипуляций реляционными данными. Семантики транзакций, постоянство значений и долговечность хранения данных являются гарантиями, требуемыми от систем баз данных такими организациями, как банки. Транзакции гарантируют, что будут записаны либо все данные, либо ничего при комбинировании нескольких потенциально сложных операций в рамках одной, аналогичной списанию денежных средств с одного счета и переводу их на другой счет операции. Постоянство значений заключается в том, что все выполняемые после обновления значения запросы будут возвращать обновленное значение. Долговечность хранения данных гарантирует то, что как только значение подвергается изменению, оно сохраняется в постоянном хранилище (таком, как жесткий диск) и может быть восстановлено в случае краха базы данных.

Системы NoSQL отказываются от некоторых из этих гарантий в обмен на улучшение производительности, что позволяет добиться допустимого и предсказуемого поведения некоторых, не имеющих отношения к банковским системам приложений. Эти ослабления гарантий вместе с изменениями модели данных и языка запросов обычно упрощают безопасное распределение базы данных по множеству машин в случае увеличения объема хранимых данных до пределов возможностей системы хранения данных отдельной машины.

Системы NoSQL находятся в большей степени на раннем этапе своего развития. Архитектурные решения описанных в данной главе систем являются реализациями требований различных пользователей. Наиболее сложной задачей при обобщении архитектурных возможностей некоторых проектов с открытым исходным кодом является то обстоятельство, что каждый проект является движущейся мишенью. Помните о том, что подробности реализации отдельных систем будут меняться. Когда вы будете выбирать систему NoSQL, вы можете использовать эту главу в качестве руководства, но не для выбора системы на основе реализуемых возможностей.

При размышлениях о системах NoSQL следует определиться с приведенными ниже вопросами:

- *Модель данных и запросов:* Представлены ли ваши данные в виде строк, объектов, структур данных или документов? Можете ли вы осуществлять запрос к базе данных, предусматривающий расчет с объединением множества записей?
- *Долговечность:* При изменении значения следует ли немедленно сохранять его в постоянном хранилище? Должно ли оно храниться на нескольких машинах для восстановления в случае потери данных на одной из них?
- *Масштабируемость:* Могут ли ваши данные размещаться на одном сервере? Обуславливает ли количество операций чтения и записи необходимость использования множества дисков для работы под нагрузкой.
- *Распределение:* Необходимо ли хранить данные на нескольких серверах для масштабирования системы, доступности данных, а также долговечности их хранения? Как вы узнаете о том, какая запись находится на каком из серверов?
- *Постоянство:* Если вы разделили данные и скопировали их на множество серверов, то как серверы будут взаимодействовать друг с другом при изменении записи?
- *Семантики транзакций:* В случаях, когда вы выполняете последовательности операций, некоторые базы данных позволяют объединить их в транзакцию, которая предоставляет некоторое подмножество гарантий ACID (Атомарность (Atomic), Постоянство (Consistency), Изоляция (Isolation) и Долговечность (Durability)) для текущей и уже выполняемых транзакций. Требует ли ваша бизнес-логика этих гарантий, обычно приводящих к потерям производительности?

- *Производительность отдельного сервера:* Если вы хотите безопасно хранить данные на диске, какие структуры данных на диске наиболее оптимальны для нагрузок с большим количеством операций чтения и записи? Является ли операция записи на диск причиной снижения производительности?
- *Анализ нагрузок:* Мы будем в значительной степени обращать внимание на нагрузки, связанные с поиском данных в случаях необходимости запуска отзывчивого пользовательского веб-приложения. В ряде случаев вам может понадобиться создавать сопоставимые по размерам с хранящимися данными отчеты, например, с целью сбора статистики в отношении множества пользователей. Требуются ли в вашем случае для вашего набора инструментов подобные функции?

Хотя мы рассмотрим все эти вопросы, последние три одинаково важных вопроса будут обсуждаться в меньшей степени.

13.2. Модели данных и запросов систем NoSQL

Модель данных базы данных задает то, как данные будут логически организованы. Ее модель запросов устанавливает способ получения и обновления данных. Стандартными моделями данных являются реляционная модель, модель хранилища с доступом на основе ключей или различные модели на основе графов. Языки запросов, о которых вы могли слышать, включают SQL, поиск на основе ключей и MapReduce. Системы NoSQL комбинируют различные модели данных и запросов, что в итоге приводит к появлению различных архитектурных решений.

13.2.1. Модели данных систем NoSQL на основе ключей

Системы NoSQL обычно исключают использование реляционной модели и всей выразительности языка SQL, ограничивая поиск в наборах данных одним полем. Например, даже если у записи, соответствующей работнику имеется ряд свойств, у вас есть возможность получить только запись работника с помощью идентификатора этой записи. В результате большинство запросов в системах NoSQL представляют собой поиск на основе ключа. Разработчик выбирает ключ, идентифицирующий каждую из записей и может, в большинстве случаев, извлечь только записи, выполняя поиск их ключей в базе данных.

В системах, основанных на поиске ключей, сложные объединения операций или извлечение данных, соответствующих нескольким ключам, могут потребовать нестандартного подхода к использованию имен ключей. Разработчик, желающий найти запись работника по ее идентификатору и найти всех работников отдела, может создать два типа ключей. Например, ключ `employee:30` будет указывать на запись работника с идентификатором 30, а ключ `employee_departments:20` может указывать на список всех работников отдела с идентификатором 20. Операция объединения запросов переносится в область логики приложения: для получения записей работников отдела 20 приложение сначала получает список идентификаторов работников с помощью ключа `employee_departments:20`, после чего в цикле получает записи работников с помощью ключей `employee:ID` с использованием списка идентификаторов работников.

Модель поиска ключей выгодно отличается благодаря тому, что она предполагает использование постоянных шаблонов запросов - общая нагрузка состоит из нагрузок за счет поиска ключей, которые относительно однообразны и предсказуемы. Профилирование с целью поиска медленно исполняющихся участков кода приложения проще, так как все сложные операции реализованы в коде приложения. С другой стороны, логика работы с моделью данных и бизнес-логика становятся более взаимосвязанными, что затрудняет процесс создания абстракций.

Давайте кратко затронем типы данных, ассоциированных с каждым ключом. Различные системы NoSQL предлагают различные решения в данной области.

Хранилища пар ключ-значение

Простейшей формой хранилища системы NoSQL является хранилище пар ключ-значение. Каждый ключ ставится в соответствие значению, в форме произвольных данных. Хранилище системы NoSQL не располагает информацией об этих данных и просто передает их приложению. В нашем

примере базы данных работников Employee ключу `employee:30` могут соответствовать данные в формате JSON или таких бинарных форматах, как Protocol Buffers⁴, Thrift⁵ или Avro⁶, хранящие информацию о работнике с идентификатором 30.

Если разработчик использует структурированные форматы для хранения сложных структур данных, соответствующих ключу, он должен обрабатывать данные на уровне приложения: хранилище пар ключ-значение в общем случае не предоставляет механизмов для запросов ключей на основании некоторых свойств соответствующих им значений. Хранилища пар ключ-значение отличаются простотой их модели запросов, обычно состоящей из примитивов для установки, получения и удаления значений (`set`, `get` и `delete`), но не предусматривают возможности добавления простых функций фильтрации на уровне базы данных ввиду непрозрачности этих значений. Система Voldemort, основанная на системе Dynamo компании Amazon, предоставляет распределенное хранилище пар ключ-значение. Система BDB⁷ предоставляет библиотеку, реализующую интерфейс для работы с парами ключ-значение.

Хранилища пар ключ-структура данных

Хранилища пар ключ-структура данных, ставшие популярными благодаря системе Redis⁸, ассоциируют каждое значение с определенным типом. В Redis доступные типы, которые могут использоваться значением, включают в себя: целочисленные значения, строки, списки, множества и отсортированные множества. В дополнение к примитивам `set` / `get` / `delete` существуют такие специфичные для типов команды, как увеличение уменьшение целочисленного значения или добавление/извлечение элементов списка, позволяющие реализовать функции запросов модели без значительного влияния на характеристики производительности. Предоставляя простые специфические для типов функции, при этом избегая таких операций как сборки или объединения при работе с множеством ключей, система Redis поддерживает баланс между функциональностью и производительностью.

Хранилища пар ключ-документ

Хранилища пар ключ-документ, такие, как CouchDB⁹, MongoDB¹⁰ и Riak¹¹ ставят в соответствие ключу какой-либо документ, содержащий структурированную информацию. Эти системы хранят документы в JSON или подобном JSON формате. Они содержат списки и словари, которые могут рекурсивно встраиваться друг в друга.

Система MongoDB разделяет пространство ключей на коллекции, поэтому ключи для записей работников (Employees) и записей отделов (Department), например, не будут пересекаться. Системы CouchDB и Riak возлагают задачу отслеживания типов на разработчика. Свобода действий и сложность хранилищ документов представляют обоюдоострый меч: разработчики приложений получают свободу моделирования своих документов, но логика запросов в рамках приложения может стать чрезмерно сложной.

Хранилища семейств столбцов BigTable

Системы HBase и Cassandra основывают свои модели данных на модели, используемой системой BigTable компании Google. В этой модели ключ идентифицирует строку, которая содержит данные, хранящиеся в одном или нескольких семействах столбцов (Column Families - CF). В рамках семейства столбцов каждая строка может содержать множество столбцов. Значения в каждом столбце содержат метку времени, поэтому несколько версий соответствий между строкой и столбцом могут находиться в одном семействе столбцов.

Концептуально можно рассматривать семейства столбцов как хранилища ключей сложной формы (идентификатор строки, семейство столбцов, столбец, метка времени), соответствующих значениям, отсортированным на основе их ключей. В результате работы по проектированию данной системы были выработаны архитектурные решения в области моделей данных, позволившие перене-

сти большую часть функций в пространство ключей. Это особенно полезно при моделировании данных истории с метками времени. Модель изначально поддерживает распределенное размещение столбцов, так как идентификаторы строк, не содержащие необходимых столбцов не должны явно указывать на значения NULL для этих столбцов. С другой стороны, столбцы, содержащие несколько значений NULL или вообще не содержащие их, все же должны хранить идентификатор столбца для каждой строки, что ведет к еще большему потреблению дискового пространства.

Модель данных каждого проекта так или иначе отличается от оригинальной модели системы BigTable, но в рамках системы Cassandra она претерпела наиболее значительные изменения. Система Cassandra вводит понятие суперстолбца в рамках каждого семейства столбцов для реализации нового уровня соответствия, моделирования и индексирования. Она также устраняет понятие локальных групп, которые могли физически хранить объединенное множество семейств столбцов для повышения производительности.

13.2.2. Хранилища на основе графов

Одним из классов хранилищ систем NoSQL являются хранилища на основе графов. Не все генерируемые данные аналогичны по своей структуре и реляционная модель, а также модели на основе ключей не всегда лучшим образом подходят для хранения и осуществления запросов любых данных. Графы являются фундаментальной структурой в компьютерных науках и системы HyperGraphDB¹² и Neo4J¹³ являются двумя популярными системами NoSQL для хранения данных, структурированных в форме графов. Хранилища на основе графов практически по всем параметрам отличаются от других типов хранилищ, рассмотренных нами ранее: они используют отличные модели данных, шаблоны для обхода данных и создания запросов, физическое размещение данных на диске, метод распределения данных по множеству машин, а также семантики запросов транзакций. Мы не будем рассматривать эти фундаментальные отличия из-за ограничений объема главы, но вы должны понимать, что определенные классы данных могут более успешно храниться и модифицироваться с помощью запросов в случае применения хранилищ на основе графов.

13.2.3. Сложные запросы

Существуют известные исключения в отношении методов поиска данных в системах NoSQL только на основе ключей. Система MongoDB позволяет индексировать ваши данные на основе любого количества свойств и использует язык относительно высокого уровня для указания того, какие данные вы хотели бы извлечь. Системы на основе BigTable поддерживают сканеры для итерации в рамках семейства столбцов и выбора определенных элементов с помощью фильтрации по столбцам. Система CouchDB позволяет создавать различные отображения данных и выполнять задачи MapReduce в отношении вашей таблицы для упрощения выполнения более сложных операций поиска и обновления данных. Большинство систем имеют биндинги для системы Hadoop или другого фреймворка MapReduce для выполнения аналитических запросов в отношении наборов данных.

13.2.4. Транзакции

Системы NoSQL обычно отдают приоритет производительности системы над семантиками транзакций. Другие системы на основе SQL позволяют использовать любой набор выражений в рамках транзакции - от простых выборок строк на основе первичного ключа до сложных объединений нескольких таблиц, которые впоследствии используются для усреднения значений нескольких полей.

Эти базы данных на основе SQL предоставляют гарантии ACID для транзакций. Выполнение множества операций в рамках транзакции атомарно (Atomic, буква A в аббревиатуре ACID), что означает выполнение либо всех операций, либо отказ от их выполнения. Постоянство (Consistency, буква C) подразумевает уверенность в том, что после выполнения транзакции база данных будет находиться в обычном состоянии и не будет повреждена. Изоляция (Isolation, буква I) обозначает уверенность в том, что если две транзакции будут работать с одной и той же записью, они не бу-

дут мешать друг другу. Долговечность (Durability, буква D, данная тема будет подробно освещена в следующем разделе) подразумевает уверенность в том, что как только транзакция будет завершена, измененные данные будут сохранены в надежном месте.

Предоставляющие гарантии ACID транзакции облегчают жизнь разработчикам, упрощая процесс выяснения состояния их данных. Представьте множество транзакций, каждая из которых состоит из множества шагов (т.е. сначала проверяется состояние банковского счета, затем с него списывается \$60, после чего сумма обновляется). Предоставляющие гарантии ACID базы данных обычно ограничены в возможности изменения последовательности этих шагов с учетом необходимости предоставления корректного результата в ходе всех транзакций. Это требование корректности приводит к обычно неожиданным характеристикам производительности, причем медленная транзакция может перевести быструю транзакцию в состояние ожидания выполнения.

Большинство систем NoSQL рассматривает производительность как более важный аспект, нежели гарантии ACID, но предоставляет гарантии на уровне ключей: две операции с одним и тем же ключом будут выполнены последовательно для предотвращения серьезного повреждения пар ключ-значение. Для многих приложений это решение не вызовет серьезных проблем с корректностью работы и позволит чаще выполнять быстрые операции. Однако, такой подход требует большего количества решений в области архитектуры приложения и корректности его работы со стороны разработчика.

Система Redis является известным исключением из общей тенденции отказа от транзакций. При работе на одном сервере, она предоставляет команду `MULTI` для атомарного и постоянного комбинирования множества операций, а также команду `WATCH` для изоляции операций. Другие системы предоставляют низкоуровневые функции проверки и установки значения (`test-and-set`), реализующие в некоторой степени гарантии изоляции.

13.2.5. Хранилище данных без жестко заданной схемы

Одним из распространенных параметров многих систем NoSQL является отсутствие жесткого требования к использованию схем в базе данных. Даже в хранилищах документов и хранилищах, использующих семейства столбцов, свойства подобных объектов не обязаны быть одинаковыми. Этот подход имеет преимущество, заключающееся в требовании меньшего структурирования данных и меньших затрат производительности при модификации схем в реальном времени. Данное решение возлагает большую ответственность на разработчика приложений, который должен использовать более безопасные методы программирования. Например, является ли отсутствие свойства `lastname` с фамилией в записи работника ошибкой, которую нужно исправить, или вызвано обновлением схемы, которая в данный момент используется системой? Управление данными и схемами реализуется на уровне стандартного кода приложения после нескольких итераций проекта, работающего с системами NoSQL, поддерживающими хранилища без жестко установленных схем.

13.3. Долговечность хранения данных

В идеальном случае модификации данных в системе для их хранения должны быть незамедлительно сохранены и скопированы на множество узлов для предотвращения потерь данных. Однако, поддержка механизмов безопасного хранения данных противоречит высокой производительности, поэтому различные системы NoSQL предоставляют *различные гарантии* в отношении долговечности хранения данных для повышения производительности. Сценарии неполадок достаточно многочисленны и значительно отличаются друг от друга, при этом не все системы NoSQL в состоянии защитить вас от них.

Простым и часто встречающимся сценарием неполадки является перезагрузка сервера или отключение энергоснабжения. Долговечность хранения данных в такой ситуации зависит от того, сколько

пированы ли данные из оперативной памяти на жесткий диск, которому не требуется питания для их хранения. На случай отказа жесткого диска данные копируются на сторонние устройства, которые могут быть другими жесткими дисками этой же машины (зеркалирование RAID) или другими машинами в сети. Однако, централизованный доступ к данным может не пережить ситуации, которая приводит к различным неполадкам (например, торнадо), поэтому некоторые организации доходят до того, что копируют данные на серверы центров хранения, находящихся на расстоянии нескольких фронтов ураганов. Запись данных на жесткие диски и копирование данных на множество локальных серверов или серверов в удаленных центрах обходится достаточно дорого, поэтому различные системы NoSQL меняют гарантии долговечности хранения данных на возможность повышения производительности.

13.3.1. Долговечность хранения данных на отдельном сервере

Простейшей технологией длительного хранения данных является технология длительного хранения данных на отдельном сервере, которая позволяет быть уверенными в том, что любые модификации данных будут доступны после перезагрузки сервера или отключения энергоснабжения. Обычно эта технология предполагает запись измененных данных на диск, что является слабым местом при работе под нагрузкой. Даже если вы используете функцию вашей операционной системы для записи данных в файл на диске, операционная система может добавить данные в буфер, избегая незамедлительной модификации файла на диске таким образом, что несколько операций записи могут быть сгруппированы в одну. Только осуществление системного вызова `fsync` заставит операционную систему предпринять попытку записи и удостовериться в том, что буферизованные данные модификаций находятся на диске.

Стандартные жесткие диски могут выполнить 100-200 операций случайного доступа (переходов) в секунду и ограничены скоростью последовательной записи, равной 30-100 МБ/с. Операции в оперативной памяти могут быть на порядки быстрее в обоих случаях. Эффективная технология длительного хранения данных подразумевает ограничение количества случайных операций записи, осуществляемых в вашей системе, и повышение количества последовательных операций записи на жесткий диск. В идеальном случае вы пожелаете добиться от системы минимизации количества операций записи между вызовами `fsync`, а также максимизации количества последовательных записей, и предпочтете никогда не сообщать пользователю о том, что его данные были успешно записаны на диск до момента осуществления записи с помощью вызова `fsync`. Давайте рассмотрим несколько техник повышения производительности операций, необходимых для реализации гарантий длительного хранения данных на отдельном сервере.

Контроль частоты использования вызова `fsync`

Система Memcached^{[14](#)} является примером системы, не предоставляющей гарантий длительного хранения данных, при этом использующей возможность осуществления чрезвычайно быстрых операций в пределах оперативной памяти. При перезагрузке сервера данные на этом сервере пропадают: это подходит для кэширования, но не подходит для длительного хранения данных.

Система Redis предоставляет разработчикам возможность изменения параметров использования системного вызова `fsync`. Разработчики могут установить принудительный вызов `fsync` после каждой операции записи, что является медленным и опасным решением. Для лучшей производительности система Redis может вызывать `fsync` через каждые N секунд. При самом плохом сценарии вы потеряете данные, записанные в ходе осуществления операций в течение последних N секунд, что может быть допустимо для определенных случаев использования системы. Наконец, для случаев использования системы, в которых долговечность хранения данных не важна (сбор неточной статистики или использование системы Redis для кэширования) разработчик может полностью отключить использование системного вызова `fsync`: операционная система в конечном счете все равно запишет данные на диск, но невозможно гарантировать то, что данные будут записаны в определенный момент.

Увеличение количества последовательных записей с помощью журналирования

Некоторые структуры данных, такие, как бинарные деревья (B+Trees) позволяют системам NoSQL быстро извлекать данные с диска. Обновления этих структур приводят к обновлению случайных участков файлов, хранящих структуры данных, что в итоге приводит к нескольким случайным записям при обновлении данных в случае использования вызова `fsync` после каждого обновления. Для снижения количества случайных записей такие системы, как Cassandra, Hbase, Redis и Riak добавляют данные операций обновления в последовательно записываемый файл, называемый *журналом*. В то время, как при записи других используемых системой структур данных вызов `fsync` используется периодически, при записи данных в журнал вызов `fsync` используется постоянно. Рассматривая журнал как отправную точку для восстановления состояния базы данных после краха, эти хранилища способны превращать случайные обновления данных в последовательные.

Хотя такие системы NoSQL, как MongoDB и выполняют непосредственную запись структур данных, другие системы еще больше развиваются идею использования журнала. Системы Cassandra и HBase используют заимствованную из системы BigTable технику объединения журнала и структур для поиска в рамках одного дерева слияния со структурой журнала (log-structured merge tree). Система Riak предоставляет аналогичные функции в рамках хэш-таблицы со структурой журнала (log-structured hash table). Система CouchDB использует модификацию традиционного бинарного дерева (B+Tree) так, что все изменения структуры данных добавляются к структуре на физическом устройстве хранения. С помощью этих техник удается достичь повышенной пропускной способности, но становится необходимым постоянное уплотнение данных журнала с целью предотвращения неконтролируемого роста его размера.

Повышение пропускной способности путем группировки операций записи

Система Cassandra группирует множество параллельных обновлений данных, выполненных в течение короткого промежутка времени, для единственного вызова `fsync` после их записи. Это архитектурное решение, называемое *групповой записью* (group commit) приводит к задержке при обновлении данных, так как пользователям приходится ждать выполнения нескольких параллельных обновлений данных для того, чтобы их обновление получило подтверждение. Повышение задержки позволяет повысить пропускную способность, так как множество записей в журнал может быть произведено с использованием единственного вызова `fsync`. Если рассматривать запись, то каждое обновление данных системой HBase производится с использованием низкоуровневого хранилища, предоставляемого распределенной файловой системой Hadoop (Hadoop Distributed File System - HDFS)¹⁵, для которой недавно были применены патчи, реализующие поддержку дополнения файлов с учетом использования вызова `fsync` и групповую запись.

13.3.2. Долговечность хранения данных на множестве серверов

Так как жесткие диски и компьютеры обычно после выхода из строя не подлежат восстановлению, копирование важных данных на сторонние компьютеры необходимо. Многие системы NoSQL предоставляют функции для хранения данных на множестве серверов.

Система Redis использует традиционный подход с ведущим и ведомыми серверами для копирования данных. Все операции, выполняемые с ведущим сервером, передаются в подобном журнале виде ведомым серверам, которые повторяют операции, используя свое собственное аппаратное обеспечение. Если ведущий сервер отказывает, ведомый сервер может начать работу, используя полученные от ведущего сервера данные состояния из журнала операций. Эта конфигурация может привести к потере некоторого объема данных, так как ведущий сервер не проверяет, записал ли данные в журнал ведомый сервер перед отправкой подтверждения выполнения операции пользователю. Система CouchDB использует упрощенный вариант аналогичной системы направленной репликации, в которой серверы могут быть настроены таким образом, что изменения документов будут копироваться в сторонние хранилища данных.

Система MongoDB предоставляет механизм наборов копий, в котором некоторое количество серверов ответственно за хранение каждого документа. MongoDB предоставляет разработчикам возможность проверять, обновляются ли все копии, или продолжать работу без проверки обновления всех копий с использованием новейших данных. Множество других распределенных хранилищ систем NoSQL поддерживает возможность копирования данных на множество серверов. Система HBase, основанная на HDFS, распределяет данные между серверами средствами файловой системы HDFS. Все операции записи повторяются двумя или большим количеством серверов HDFS перед возвращением управления пользователю, что позволяет быть уверенным в долговечности хранения данных на множестве серверов.

Системы Riak, Cassandra и Voldemort поддерживают более настраиваемые формы репликации. При наличии небольших отличий, все три системы позволяют пользователю задать значение параметра n , определяющего количество машин, которые в конечном итоге должны хранить копию данных, а также параметр $w < n$, который устанавливает количество машин, которые должны убеждаться в завершении записи перед возвращением управления пользователю.

Для преодоления ситуации, при которой весь датацентр прекращает работу, необходима возможность копирования данных на серверы других датацентров. Системы Cassandra, HBase и Voldemort поддерживают *независимые от стоеч* (rack aware) конфигурации, позволяющие задавать стойку или датацентр, в которых расположены различные машины. В общем случае, блокировка действий пользователя до того момента, как удаленный сервер подтвердит факт обновления данных, приводит к появлению очень большой задержки. Данные для обновлений передаются без подтверждений при работе в сетях большого размера и сохранении резервных копий данных в отдельных датацентрах.

13.4. Масштабирование с целью повышения производительности

Только что обсудив работу в условиях неполадок, давайте представим более приятную ситуацию: успешное функционирование системы! Если созданная вами система успешно функционирует, ваше хранилище данных станет одним из компонентов системы, который снизит производительность под нагрузкой. Простым и не самым элегантным решением этой проблемы является *расширение возможностей* (scale up) существующего оборудования: вы можете вложить больше средств в оперативную память и жесткие диски для обработки нагрузок с использованием единственной машины. После этого успешного шага денежные вливания в более дорогое аппаратное обеспечение станут невозможными. В таком случае вам придется копировать данные и распространять запросы между несколькими машинами для распределения нагрузки. Этот подход называется *распределением возможностей* (scale out) и оценивается с помощью *степени горизонтального масштабирования* (horizontal scalability) вашей системы.

Идеальной степенью горизонтального масштабирования является *линейная масштабируемость* (linear scalability), при достижении которой удвоение количества машин, задействованных в вашей системе хранения данных, удваивает количество запросов, которое система в состоянии обработать. Ключевым фактором для достижения такой степени горизонтального масштабирования является способ распределения данных между серверами. Фрагментация системы является действием, направленным на разделение нагрузки чтения и записи данных между множеством машин с целью распределения возможностей вашей системы хранения данных. Фрагментация системы является фундаментальным понятием в рамках архитектур многих систем, а именно: Cassandra, HBase, Voldemort и Riak, а недавно также MongoDB и Redis. Некоторые проекты, такие, как CouchDB созданы с учетом производительности отдельного сервера и не предоставляют встроенных решений для фрагментации системы, но сторонние проекты предоставляют программные компоненты для координации, позволяющие распределить нагрузку между независимыми установками этой системы на множестве машин.

Давайте рассмотрим несколько взаимозаменяемых терминов, с которыми вы можете столкнуться. Мы будем использовать термины *фрагментация системы* и *разделение системы* для обозначения аналогичных действий. Термины *машина*, *сервер* или *узел* относятся к какому-либо физическому компьютеру, хранящему часть разделенных данных. Наконец, *клuster* или *кольцо* относятся к группе машин, которые участвуют в работе вашей системы хранения данных.

Фрагментация системы подразумевает то, что ни одна машина не должна обрабатывать нагрузку, созданную операциями записи всего набора данных, а также ни одна машина не может ответить на запрос всего набора данных. Большинство систем NoSQL использует ключи и в моделях данных и в моделях запросов, при этом очень малое количество запросов осуществляет доступ ко всему набору данных в любом случае. Так как основной метод доступа к данным, хранящимся этими системами, связан с использованием ключей, фрагментация системы также обычно реализуется на основе ключей: с помощью некоторой функции, принимающей ключ в качестве исходных данных, устанавливается машина, на которой будет храниться пара ключ-значение. Мы рассмотрим два метода создания сопоставления ключ-машина: метод хэш-разделения и метод разделения на основе диапазонов.

13.4.1. Не фрагментируйте систему без необходимости

Фрагментация усложняет систему и, если это возможно, вы должны избегать ее. Давайте рассмотрим два способа масштабирования без фрагментации: копирование читаемых данных (read replicas) и кэширование (caching).

Копирование читаемых данных

Многие системы хранения данных принимают большее количество запросов на чтение данных, чем на их запись. Простым решением в данном случае является копирование данных на множество машин. Все запросы записи все также будут отправляться ведущему серверу. Запросы чтения же будут отправляться машинам, хранящим копии данных, обычно немного устаревших по отношению к данным на ведущем сервере.

Если вы уже копируете ваши данные для длительного хранения на множестве серверов и используете конфигурацию ведущих и ведомых серверов, что типично для систем Redis, CouchDB или MongoDB, ведомые машины для обслуживания запросов чтения могут взять на себя часть нагрузки ведущего сервера. Некоторые запросы, такие, как запросы для создания отчетов о ваших наборах данных, которые могут быть требовательны к ресурсам и обычно не нуждаются в обновленной с точностью до секунды копии данных, могут быть выполнены в отношении копий данных на ведомых серверах. Как правило, чем менее жесткие требования вы предъявляете к актуальности данных, тем большее количество работы вы можете перенести на ведомые серверы, тем самым повысив производительность запросов чтения данных.

Кэширование

Кэширование наиболее популярных данных в вашей системе обычно работает на удивление хорошо. Система Memcached выделяет блоки памяти на множестве серверов для кэширования данных из вашего локального хранилища данных. Клиенты Memcached пользуются преимуществами нескольких приемов горизонтального масштабирования для разделения нагрузки между установками системы Memcached на различных серверах. Для добавления памяти в пул кэширования следует просто добавить другой узел с запущенной копией системы Memcached.

Так как система Memcached проектировалась для кэширования данных, она не так сложна в плане архитектуры, как решения для постоянного хранения данных, масштабируемые в зависимости от нагрузок. Перед рассмотрением более сложных решений подумайте о том, сможет ли кэширование поспособствовать решению ваших проблем с масштабированием. Кэширование не является

исключительно временной мерой: компания Facebook использует систему Memcached для работы с объемами оперативной памяти в десятки терабайт!

Копирование читаемых данных и кэширование позволяют вам масштабировать высокие нагрузки, создаваемые запросами чтения данных. Однако, когда вы начнете повышать частоту использования операций записи и обновления ваших данных, вы будете также повышать нагрузку на ведущий сервер, хранящий все обновленные данные. В оставшейся части данного раздела мы рассмотрим техники фрагментации нагрузки, создаваемой операциями записи, с использованием множества серверов.

13.4.2. Фрагментация системы с помощью программ для координации запросов

Проект CouchDB развивается в направлении функционирования на единственном сервере. Два проекта, Lounge и BigCouch, упрощают операцию фрагментации нагрузок на систему CouchDB с помощью внешнего прокси-сервера, который работает как система предварительной обработки запросов, передаваемых функционирующими системами CouchDB. В данной архитектуре отдельные установки системы не подозревают о существовании друг друга. Программа координации распределяет запросы между отдельными установками системы CouchDB на основе ключей запрашиваемых документов.

Компания Twitter встроила механизмы фрагментации нагрузок и копирования данных с фреймворк для координации с названием Gizzard. Фреймворк Gizzard использует отдельные хранилища данных любых типов - вы можете использовать слой совместимости системами хранения данных SQL и NoSQL - и объединяет их в деревья любой глубины для разделения ключей на основе диапазонов ключей. Для снижения восприимчивости к неполадкам фреймворк Gizzard¹⁶ может быть настроен таким образом, что данные для одного и того же диапазона ключей будут копироваться на множество физических машин.

13.4.3. Последовательные хэш-кольца

Качественные хэш-функции распределяют набор ключей равномерно. Это делает их мощным инструментом для распределения пар ключ-значение между множеством серверов. В научной технической литературе подробно описана техника под названием "*"последовательное хэширование"*" (consistent hashing), которая впервые была применена для организации хранилищ данных в рамках систем с названием "*"распределенные хэш-таблицы"*" (distributed hash tables - DHTs). Системы NoSQL, построенные на принципах системы Dynamo от компании Amazon применяют данную технику распределения, а также она используется в системах Cassandra, Voldemort и Riak.

Хэш-кольца в примере

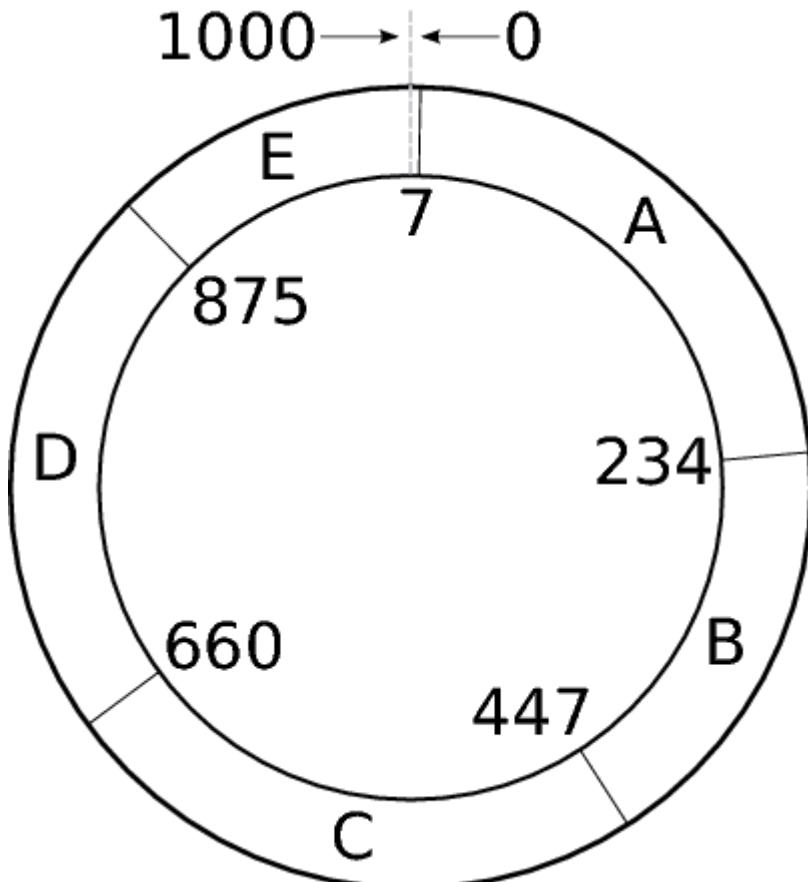


Рисунок 13.1: Кольцо распределенной хэш-таблицы

Последовательные хэш-кольца работают следующим образом. Представим, что мы используем хэш-функцию h , которая производит равномерное распределение ключей, ставя их в соответствие большим целочисленным значениям. Мы можем сформировать кольцо из чисел в диапазоне $[1, L]$, которое замыкается и может указать на позицию числа, полученного с использованием функции $H(\text{ключ}) \bmod L$ для какого-либо относительно большого целочисленного значения L . С помощью данной функции любой ключ будет поставлен в соответствие значению из диапазона $[1, L]$. Последовательное хэш-кольцо для серверов формируется путем получения уникального идентификатора каждого сервера (например, его IP-адреса) и применения к нему функции H . Вы можете интуитивно понять принцип работы данного алгоритма, обратившись к сформированному из пяти серверов (A-E) хэш-кольцу на [Рисунке 13.1](#).

В данном случае мы выбираем значение $L=1000$. Давайте представим, что $h(A) \bmod L = 7$, $h(B) \bmod L = 234$, $h(C) \bmod L = 447$, $h(D) \bmod L = 660$ и $h(E) \bmod L = 875$. Теперь мы можем сказать, на каком сервере должен находиться ключ. Для этого мы поставим в соответствие всем ключам серверы, определяя, попадает ли ключ в диапазон значений, соответствующих каждому из серверов. Например, сервер A отвечает за хранение ключей, хэш-значения которых находятся в диапазоне $[7, 233]$, а E отвечает за хранение ключей с хэш-значениями в диапазоне $[875, 6]$ (этот диапазон пересекает значение 1000). Таким образом, если $h('employee30') \bmod L = 899$, то этот ключ будет храниться на сервере E, а если $h('employee31') \bmod L = 234$, то ключ будет храниться на сервере B.

Копирование данных

Копирование для длительного хранения данных на множество серверов осуществляется путем передачи ключей и значений из одного ассоциированного с сервером диапазона серверам, следующим далее в кольце. Например, при трехкратной репликации ключи, соответствующие диапазону $[7, 233]$, будут храниться на серверах A, B и C. Если сервер A прекратит работу из-за неполадок, соседние серверы B и C примут на себя предназначенную для этого сервера нагрузку. Некоторые архитектуры предусматривают возможность временного копирования данных на сервер E и переноса

на него нагрузки сервера A, после чего диапазон значений сервера E будет расширен для включения в него значений, раньше соответствующих серверу A.

Достижение лучшего распределения

Хотя хэширование и является статистически эффективным методом равномерного распределения пространства ключей, обычно требуется большое количество серверов для того, чтобы это распределение стало действительно равномерным. К сожалению, мы обычно начинаем работу с малого количества серверов, которые не идеально отделяются друг от друга с помощью данной хэш-функции. В нашем примере длина диапазона ключей сервера A равна 277 и в то же время длина диапазона ключей сервера E равна 132. Это обстоятельство ведет к неравномерной нагрузке на серверы. Также оно осложняет процесс переноса функций с одного сервера на другой в случае неполадок, так как соседний сервер может внезапно получить контроль над всем диапазоном вышедшего из строя сервера.

Для решения проблемы неравномерности диапазонов ключей многие системы, включая Riak, создают по нескольку "виртуальных" узлов на физической машине. Например, при наличии 4 виртуальных узлов сервер A будет функционировать как серверы A_1, A_2, A_3 и A_4. Каждый виртуальный узел использует для хэширования различные значения, увеличивая вероятность управления ключами, распределенными по различным частям пространства ключей. Система Voldemort использует аналогичный подход, при котором количество диапазонов значений настраивается вручную и обычно больше количества серверов, в результате каждый сервер получает несколько небольших диапазонов значений.

Система Cassandra не ставит в соответствие каждому серверу множество небольших диапазонов, что иногда приводит к неравномерному распределению диапазонов ключей. Для балансировки нагрузки в Cassandra используется асинхронный процесс, который динамически определяет расположение серверов на кольце в зависимости от нагрузки на них в течение прошедшего времени.

13.4.4. Распределение с использованием диапазонов

В случае применения техники распределения с использованием диапазонов для фрагментации нагрузки некоторые машины, обслуживающие вашу систему, хранят метаданные с указанием на то, какие серверы хранят те или иные диапазоны ключей. Эти метаданные используются для поиска расположения ключа и поиска диапазонов, соответствующих серверам. Аналогично подходу с использованием последовательных хэш-колец, при распределении с использованием диапазонов пространство ключей разделяется на диапазоны, причем каждый диапазон ключей управляет одной машиной и возможно копируется на другие. В отличие от подхода с использованием последовательных хэш-колец, два ключа, находящиеся друг рядом с другом после сортировки, скорее всего, окажутся в одном и том же диапазоне. Это обстоятельство уменьшает объем метаданных для поиска диапазонов, так как большие диапазоны могут быть представлены в сжатом виде с помощью маркеров [начало, конец].

С введением активного механизма учета записей о *соответствии диапазонов серверам*, подход разделения диапазонов позволяет более точно выполнять снижение нагрузки на и без того загруженные серверы. Если в определенном диапазоне ключей фиксируется более высокий, чем в других диапазонах трафик, менеджер нагрузок может уменьшить размер диапазона для сервера или снизить количество обслуживаемых сервером фрагментов данных. Появившаяся свобода активного управления нагрузками достигается за счет введения дополнительных архитектурных компонентов для мониторинга и маршрутизации запросов.

Подход системы BigTable

Документация системы BigTable компании Google описывает иерархическую технику распределения данных в объекты (tablets) с использованием диапазонов. Объект хранит диапазон ключей и

значений строки в рамках семейства столбцов. Он осуществляет поддержку всех необходимых журналов и структур данных для ответов на запросы о ключах в соответствующем диапазоне. Серверы объектов обслуживают множество объектов в зависимости от рабочей загрузки каждого из них.

Размер каждого объекта поддерживается в пределах 100-200 МВ. Так как объекты могут изменять размер, два небольших объекта для примыкающих диапазонов ключей могут быть объединены, а также объект большого размера может быть разделен на два объекта. Ведущий сервер анализирует размер объекта, загрузку и доступность сервера объектов. Ведущий сервер устанавливает какой сервер объектов обслуживает объекты в любой момент времени.

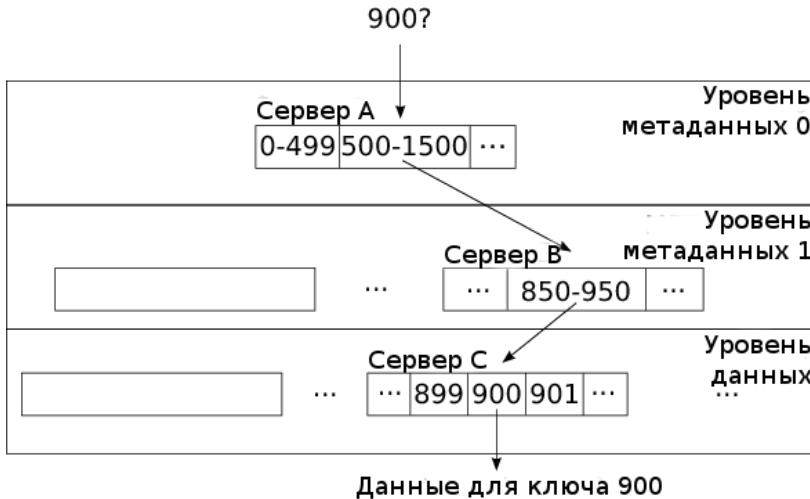


Рисунок 13.2: Распределение с использованием диапазонов системы на основе BigTable

Центральный сервер поддерживает соответствие объектов в таблице метаданных. Так как эти метаданные могут достигать больших объемов, таблица метаданных также использует фрагментацию в виде объектов, которые ставят в соответствие диапазоны ключей объектам и обозначают серверы объектов, ответственные за работу с этими диапазонами. Этот подход приводит необходимости преодоления трехслойной иерархической структуры для нахождения ключа на хранящем его сервере объектов, как показано на [Рисунке 13.2](#).

Давайте рассмотрим пример. Клиент, ищущий ключ 900 отправляет запрос серверу А, который хранит объект для метаданных уровня 0. Этот объект идентифицирует объект метаданных уровня 1 на сервере В, содержащий диапазоны ключей 500-1500. Клиент отправляет запрос серверу С с ключом, который отвечает, что объект, содержащий ключи 850-950 найден в объекте на сервере С. Наконец, клиент отправляет запрос с требованием ключа на сервер С и получает данные строки в ответ. Объекты метаданных уровней 0 и 1 могут кэшироваться клиентом, который хочет избежать создания излишней нагрузки на серверы объектов ввиду отправки повторяющихся запросов. Документация системы BigTable указывает на то, что эта трехуровневая иерархическая структура может использовать для работы 2^{61} байт полезного дискового пространства, создавая объекты размером в 128 МВ.

Обработка ошибок

Ведущий сервер является единой точкой отказа в рамках архитектуры BigTable, но он может временно прекращать работу, не влияя на запросы к серверам объектов. Если сервер объектов прекращает работу в момент обработки запросов объектов, ведущий сервер должен определить это и переназначить его объекты, при этом, запросы будут завершаться ошибкой в течение некоторого промежутка времени.

В общем случае для определения наличия и обработки неполадок машин документация BigTable рекомендует использовать Chubby, распределенную систему блокировок для управления принад-

лежностью к группам и доступностью сервера. Система ZooKeeper^{[17](#)} является реализацией Chubby с открытым исходным кодом и некоторые проекты на основе Hadoop используют ее для управления вторичными ведущими серверами и переназначения серверов объектов.

Проекты NoSQL, применяющие распределение с использованием диапазонов

Система HBase применяет иерархический подход для реализации системы распределения с использованием диапазонов BigTable. Данные объектов хранятся в распределенной файловой системе Hadoop (HDFS). Файловая система HDFS выполняет копирование данных и проверяет копии на наличие повреждений, оставляя для серверов объектов задачи по обработке запросов, обновлению структур хранилища и инициированию разделения и объединения объектов.

Система MongoDB использует технологию распределения данных с использованием диапазонов, аналогичную таковой в системе BigTable. Несколько конфигурационных узлов хранят и управляют таблицами перенаправлений, которые содержат информацию о том, какой из серверов хранения данных ответственен за какой из диапазонов ключей. Эти конфигурационные узлы синхронизируют данные с использованием протокола под названием *"двуухфазная передача"* (two-phase commit) и работают в качестве гибридного решения, состоящего из ведущего сервера BigTable для указания диапазонов и системы Chubby для управления конфигурацией в условиях высокой доступности. Отдельные процессы маршрутизации, не использующие состояния, отслеживают последние запросы изменения конфигурации маршрутизации и осуществляют перенаправления запросов к соответствующим серверам хранилища данных для получения ключей. Серверы хранилища данных распределены в группы серверов копий данных для осуществления репликации.

Система Cassandra предоставляет сохраняющую последовательность систему распределения, используемую в том случае, если желательно разрешить быстрые сканирования диапазонов с доступом к ваших данных. Серверы системы Cassandra все также объединены в кольцо с использованием последовательного хэширования, но вместо хэширования пар ключ-значение и сопоставления полученного результата с кольцом для установления сервера, которому должны соответствовать эти данные, ключ просто ставится в соответствие серверу, который контролирует диапазон значений, содержащий ключ изначально. Например, ключи 20 и 21 оба будут поставлены в соответствие серверу А в нашем последовательном хэш-кольце на [Рисунке 13.1](#), вместо хэширования и случайного распределения по кольцу.

Фреймворк Gizzard компании Twitter для управления распределенными и скопированными данными с использованием множества систем применяет распределение с использованием диапазонов для фрагментации данных. Маршрутизирующие серверы из иерархий любых глубин ставят диапазоны ключей в соответствие серверам, стоящим ниже их по иерархии. Эти серверы либо хранят данные для ключей в соответствующем им диапазоне, либо перенаправляют запросы маршрутизирующему серверу другого уровня. Репликация в данной модели реализуется с помощью отправки обновлений на множество машин для диапазона ключей. Серверы маршрутизации системы Gizzard обрабатывают ошибки записи способом, отличным от применяемых другими системами NoSQL. Система Gizzard требует, чтобы системные архитекторы сделали все обновления многократными (они могут выполняться дважды). В случае, когда узел хранилища не может выполнить запрос, узлы маршрутизации кэшируют и периодически отправляют обновления узлу до тех пор, пока он не подтвердит успешное завершение обновления данных.

13.4.5. Какую систему распределения данных следует использовать

Какую систему распределения данных предпочтительнее выбрать после рассмотрения подходов к фрагментации данных на основе хешей и диапазонов? Это зависит от условий ее использования. Распределение с использованием диапазонов является очевидным выбором в случае частого сканирования ключей для доступа к данным. Так как вы читаете данные используя ключи, вы не будете осуществлять обращения к случайным узлам сети, усиливая тем самым загрузку на нее. Но

если вам не требуется сканировать диапазоны данных, то какую схему их фрагментации следует использовать?

Разделение данных с использованием хэшей позволяет достичь разумной степени их распределения по узлам, а случайные диспропорции в распределении могут быть сглажены с помощью создания виртуальных узлов. В схеме распределения данных с помощью хэширования маршрутизация реализуется достаточно просто: в большей части случаев хэш-функция может выполняться клиентами для поиска необходимого сервера. В случае применения более сложных схем ребалансировки поиск необходимого сервера для ключа становится более сложной задачей.

Распределение с использованием диапазонов требует дополнительных затрат ресурсов на поддержку серверов маршрутизации и конфигурации, которые будут работать под большими нагрузками и станут центральными точками отказа в случае отсутствия относительно сложных схем обработки ошибок. Однако, при правильной настройке, данные, распределенные с использованием диапазонов, для балансировки нагрузки будут разделены на небольшие фрагменты, которые могут быть перемещены в другие диапазоны при высокой нагрузке на систему. В случае отключения сервера соответствующие ему диапазоны могут быть разделены между множеством других серверов вместо создания дополнительной нагрузки на его соседние серверы в период неработоспособности.

13.5. Согласованность данных

После обсуждения достоинств техник копирования данных на множество машин для повышения долговечности их хранения и распределения нагрузки, пришло время открыть вам секрет: хранение копий ваших данных на множестве машин при условии их идентичности является сложной задачей. На практике копии повреждаются и не соответствуют друг другу, теряются без возможности последующего восстановления, сети не позволяют синхронизировать наборы копий, а также передаваемые между машинами сообщения доставляются с задержкой или теряются. Существует два основных подхода к обеспечению согласованности данных в экосистеме NoSQL. Первый подход подразумевает строгую согласованность (strong consistency) данных, при которой все копии остаются синхронизированными. Второй подход подразумевает конечную согласованность (eventual consistency), при которой копии могут быть не идентичными, но в конечном счете все же стать таковыми. Для начала давайте разберемся с тем, почему второй вариант является допустимым решением, рассмотрев фундаментальное понятие систем распределенных вычислений. После этого мы перейдем к подробному рассмотрению каждого из подходов.

13.5.1. Немного о CAP

Почему мы учитываем не все гарантии строгой согласованности наших данных? Все сводится параметрам распределенных систем, спроектированных для работы с современным сетевым оборудованием. Идея была впервые предложена Eric Brewer в виде теоремы CAP (CAP Theorem), после чего была также представлена в работе от Gilbert и Lynch [[GL02](#)]. Теорема впервые описывала три параметра распределенных систем, из которых и был сформирован акроним CAP:

- *Согласованность (Consistency)*: все ли копии данных обычно логически соответствуют одной и той же версии этих данных в момент их чтения? (Этот параметр отличается от последовательности, обозначенной с помощью буквы С в аббревиатуре ACID.)
- *Доступность (Availability)*: Отправляются ли ответы на запросы чтения и записи копий данных независимо от количества недоступных копий?
- *Возможность разделения (Partition tolerance)*: Может ли система продолжить работу даже если потеряна возможность доступа к нескольким копиям посредством сети.

Теорема говорит о том, что система хранения данных, которая функционирует на множестве компьютеров может обладать только двумя из этих параметров в ущерб третьему. Также нам приходится реализовывать системы с возможностью разделения. При работе с современным оборудованием сетей с использованием современных протоколов сообщений пакеты могут теряться, свитчи

могут выходить из строя и не существует способа для получения информации о неработоспособности сети или сервера. Все системы NoSQL должны обладать возможностью разделения. Остается выбор между согласованностью и доступностью. Ни одна система NoSQL не может обладать двумя этими параметрами одновременно.

Выбор в пользу согласованности подразумевает то, что ваши копии данных не будут рассинхронизированы. Простейшим путем достижения согласованности является требование подтверждения всех обновлений, производимых в отношении копий данных. В случае недоступности копии и невозможности ее обновления, вы можете снизить доступность соответствующих ее ключам данных. Это значит, что до того момента, как все копии будут восстановлены и доступны, пользователь не получит подтверждения успешного завершения операции обновления их данных. Следовательно, выбор в пользу согласованности является выбором в пользу отсутствия круглосуточной доступности каждого элемента набора данных.

Выбор в пользу доступности подразумевает, что при выполнении операции пользователем копии должны предоставить свои данные независимо от состояния других копий. Это может привести к расхождению в согласованности данных копий, так как в данном случае не требуется подтверждения всех их обновлений и некоторые копии могут не получить всех обновлений.

Положения теоремы CAP ведут к появлению подходов строгой согласованности и конечной согласованности при разработке хранилищ данных NoSQL. Существуют и такие подходы, как облегченная согласованность и облегченная доступность, представленные в системе PNUTS [[CRS+08](#)] компании Yahoo!. Ни в одной из обсуждаемых нами систем NoSQL с открытым исходным кодом эта техника пока не реализована, поэтому мы не будем рассматривать ее более подробно.

13.5.2. Строгая согласованность данных

Системы, предусматривающие строгую согласованность данных, следят за тем, чтобы копии данных были идентичны и позволяли получить одно и то же значение ключа. Некоторые копии могут быть не синхронизированы с другими, но в момент, когда пользователь запрашивает значение для ключа `employee30:salary`, машины имеют возможность согласования значения, передаваемого пользователю. Принцип работы этого механизма лучше всего описывается с помощью чисел.

Например, мы копируем ключ на N машин. Какая-либо машина, возможно, одна из множества N, выступает в качестве координатора для каждого запроса. Координатор устанавливает факт того, что определенное количество машин из множества N принял и подтвердило каждый из запросов. В момент, когда происходит запись обновленных данных для ключа, координатор не отправляет подтверждение завершения обновления пользователю до того момента, как как будет принято подтверждение получения обновлений группой из w копий. Когда пользователь хочет прочитать значение, соответствующее какому-либо ключу, координатор отправляет ответ, когда как минимум из R копий прочитано одно и то же значение. Мы говорим, что система использует строгую согласованность данных, если $R+W>N$.

Используя числа для иллюстрации данной идеи, представим, что мы копируем каждый ключ на $N=3$ машины (обозначим их A, B и C). Предположим, что ключ `employee30:salary` имеет начальное значение \$20,000, но мы хотим повысить жалование сотрудника `employee30` до \$30,000. Давайте также установим требование, согласно которому как минимум $w=2$ машины из трех A, B или C должны отправить подтверждение выполнения каждого запроса записи ключа. Если машины A и B подтверждают выполнение запроса записи данных (`employee30:salary, $30,000`), координатор оповещает пользователя об успешном обновлении ключа `employee30:salary`. Предположим, что машина C никогда не принимала запрос записи данных для ключа `employee30:salary`, поэтому все еще содержит значение \$20,000. Когда координатор примет запрос чтения данных ключа `employee30:salary`, он отправит этот запрос всем 3 машинам:

- Если мы установим значение $R=1$ и машина С первой пришлет ответ со значением \$20,000, наш сотрудник будет не очень рад.
- Однако, если мы установим значение $R=2$, координатор получит значение, отправленное машиной С, подождет второго ответа от машины А или В, который будет конфликтовать с устаревшим значением машины С и наконец получит ответ от третьей машины, который подтвердит то, что значение \$30,000 является мнением большинства.

Таким образом, для достижения строгой согласованности данных в этом случае нам потребовалось установить значение $R=2$, следовательно, $R+W=4$.

Что же случится, если w копий не будут обновлены в ходе выполнения запроса записи данных или R копий не будут прочитаны в ходе выполнения запроса чтения данных в случае необходимости использования строго согласованного ответа? Координатор может приостановить обработку запроса до истечения времени ожидания и в конечном счете отправить пользователю сообщение об ошибке или ожидать того, что ситуация разрешится сама собой. В любом случае, система окажется не в состоянии выполнить этот запрос, по крайней мере в течение некоторого промежутка времени.

Ваш выбор значений R и w влияет на то, как много машин смогут вести себя странно до того момента, как ваша система заблокирует возможность совершения различных действий с ключом. Если вы установите необходимость подтверждения записи данных для всех ваших копий, например, то будет справедливо равенство $w=N$ и операции записи будут приостанавливаться или аварийно завершаться в случае невозможности работы с любой копией. Стандартным выбором является равенство $R+W=N+1$, соответствующее минимальным требованиям строгой согласованности данных, при этом позволяющее временные несоответствия между копиями. Многие системы, реализующие подход строгой согласованности данных, выбирают равенства $w=N$ и $R=1$, так как в этом случае не придется рассчитывать на рассинхронизацию узлов.

Система HBase использует для хранения и копирования данных HDFS, распределенную прослойку для хранения данных. HDFS предоставляет гарантии строгой согласованности данных. В HDFS запись не может завершиться успешно до тех пор, пока данные не будут скопированы во все N (обычно 2 или 3) копий, поэтому справедливо равенство $w=N$. Чтение завершится успешно в случае доступности хотя бы одной копии, поэтому справедливо равенство $R=1$. Для предотвращения снижения производительности из-за интенсивных нагрузок записи, данные передаются от пользователя узлам с копиями асинхронно в параллельном режиме. Как только получено подтверждение о том, что все копии данных доставлены, финальный шаг добавления новых данных в систему выполняется атомарно и согласованно всеми узлами с копиями данных.

13.5.3. Конечная согласованность данных

Такие системы на основе Dynamo, как Voldemort, Cassandra и Riak позволяют пользователю задать необходимые значения N , R и w , даже при условии $R+W < N$. Это значит, что пользователь может достичь и строгой и конечной согласованности данных. Если пользователь выбирает конечную согласованность данных, даже в том случае, когда разработчик стремится реализовать модель строгой согласованности данных, но значение w меньше N , существуют периоды, в течение которых копии могут быть не синхронизированы. Для реализации модели конечной согласованности данных копий эти системы применяют специальные инструменты для быстрого выявления устаревших копий. Для начала давайте рассмотрим вопрос о том, как различные системы определяют рассинхронизацию копий данных, после чего обсудим их механизмы синхронизации копий данных и наконец проясним некоторые методы для ускорения процесса синхронизации, реализованные на основе методов системы Dynamo.

Управление версиями и конфликты

Так как две копии могут содержать две различных версии значения для какого-либо ключа, механизмы контроля версий данных и поиска конфликтов очень важны. Системы на основе Dynamo

используют механизм контроля версий под названием "*"вектор времени"*" (vector clocks). Вектор времени является вектором, поставленным в соответствие каждому ключу и содержащим счетчик для каждой копии. Например, если серверы A, B и C хранят три копии какого-либо ключа, вектор времени будет состоять из трех элементов (N_A, N_B, N_C) и инициализироваться с помощью значений (0, 0, 0).

Каждый раз, когда соответствующая ключу копия данных обновляется, счетчик вектора увеличивает значение. Если сервер B модифицирует ключ, который до этого имел версию (39, 1, 5) он изменит вектор времени до (39, 2, 5). Когда данные другой копии, скажем на сервере C, обновляются с использованием сервера B, происходит сравнение вектора времени сервера B с локальным вектором времени. В случаях, если счетчики локального вектора времени имеют меньшие значения, чем счетчики, полученные с сервера B, на локальном сервере хранится устаревшая копия данных, которая может быть перезаписана с использованием копии с сервера B. Если серверы B и C имеют счетчики, значения в каждом из которых больше других, скажем (39, 2, 5) и (39, 1, 6), серверы считают, что они получили различные, возможно не взаимозаменяемые обновления и констатируют наличие конфликта.

Разрешение конфликтов

Механизмы разрешения конфликтов отличаются в различных системах. Документация системы Duplicity возлагает задачу разрешения конфликтов на использующую систему хранения данных приложение. Две версии списка покупок могут быть объединены в одну без значительной потери данных, но две версии совместно редактируемого документа могут потребовать вмешательства человека для обзора и разрешения конфликта. Система Voldemort реализует эту модель, возвращая несколько копий данных ключа отправившему запрос клиентскому приложению в случае конфликта.

Система Cassandra, которая хранит метку времени для каждого ключа, использует наиболее позднюю версию данных ключа в случае конфликта двух версий. Эта возможность исключает необходимость дополнительного взаимодействия с клиентом и упрощает API. Также данное архитектурное решение затрудняет работу в ситуациях, когда конфликтующие данные могут быть объединены специальным образом, как в нашем примере со списком покупок или при реализации распределенных счетчиков. Система Riak позволяет использовать оба подхода, предоставляемых системами Voldemort и Cassandra. Система CouchDB предлагает гибридную модель: она определяет наличие конфликта и позволяет пользователям запрашивать соответствующие ключу конфликтующие данные для ручного исправления, но самостоятельно выбирает версию для возвращения данных пользователям до разрешения конфликтов.

Исправление при чтении

Если R копий при чтении координатором содержат не конфликтующие между собой данные, координатор может безопасно возвращать данные приложению. Координатор также может отметить факт рассинхронизации нескольких копий. Документация системы Duplicity рекомендует, а системы Cassandra, Riak и Voldemort реализуют технику с названием "*"исправление при чтении"*" (read repair) для разрешения таких ситуаций. В момент, когда координатор устанавливает наличие конфликта при чтении, даже если непротиворечивое значение отправлено пользователю, координатор начинает использовать протоколы разрешения конфликтов по отношению к конфликтующим копиям. Данное действие позволяет осуществить профилактическое разрешение конфликта с помощью небольшой дополнительной работы. Версии данных копий уже отосланы координатору, поэтому быстрейшее разрешение конфликта позволит снизить количество несоответствий в данных системах.

Передача данных с использование подсказок

Системы Cassandra, Riak и Voldemort используют технику под названием "*передача данных с использованием подсказок*" (hinted handoff) для повышения производительности операций записи в ситуациях, когда узел становится временно недоступным. Если одна из копий соответствующих ключу данных не может быть обновлена в из-за отсутствия ответа сервера на запрос записи, выбирается другой узел для временного переноса операций записи на его мощности. Записанные данные, предназначавшиеся для недоступного узла, хранятся отдельно и в тот момент, когда сервер резервных копий определяет, что до этого недоступный узел снова доступен, он передает все данные доступному серверу. Документация системы Dynamo использует подход "неполного кворума" (sloppy quorum) и добавляет количество записей с использованием подсказок к значению W, отражающему требуемое количество подтверждений операций записи. Системы Cassandra и Voldemort не добавляют количество записей с использованием подсказок к значению W и считают запись неудачной в случае недостаточного количества подтверждений операций записи W на заданные изначально серверы. Техника передачи данных с использованием подсказок также полезна и для этих систем, так как она ускоряет процесс восстановления данных во время перехода недоступного узла в работоспособное состояние.

Противодействие энтропии

Когда сервер с копией данных недоступен в течение длительного промежутка времени или машина, хранящая скопированные с помощью подсказки данные для недоступного сервера также становится недоступной, копии должны синхронизироваться друг с другом. В этом случае системы Cassandra и Riak реализуют процесс, аналогичный процессу в системе Dynamo и называемый "*противодействием энтропии*" (anti-entropy). В ходе этого процесса серверы обмениваются деревьями Меркль (Merkle Trees) для идентификации их участков диапазонов ключей, которые являются рассинхронизированными. Дерево Меркль является иерархической структурой для проверки данных на основе хэшей: если хэш всего пространства ключей не одинаков для двух копий, серверы будут обмениваться хэшами всех меньших и меньших участков скопированных данных пространства ключей до того момента, как рассинхронизированные ключи будут идентифицированы. Этот подход снижает объем ненужных и в большей степени идентичных данных, передаваемых между серверами.

Техника Gossip

Наконец, с ростом распределенных систем становится все сложнее узнать то, как функционирует каждый из узлов системы. Три системы, построенные на основе Dynamo, используют для отслеживания состояния узлов старую простую технику, известную под названием "*gossip*". Периодически (каждую секунду или с подобной периодичностью) узел должен выбирать другой случайный узел, с которым он уже взаимодействовал для обмена данными о состоянии других узлов системы. С помощью этого обмена данными узлы узнают о том, какие из узлов находятся в неработоспособном состоянии, а также о том, куда перенаправить клиентов, ищущих ключ.

13.6. Заключительное слово

Экосистема NoSQL все еще находится на раннем этапе развития и многие описанные нами системы изменят свои архитектуры, технические решения и интерфейсы. Наиболее важной информацией данной главы является не информация о том, какие действия каждая из систем NoSQL может выполнять в данный момент, а о технических решениях, которые обуславливают комбинацию возможностей, предоставляемых этими системами. Технология NoSQL перекладывает большой объем работы по проектированию на разработчика приложения. Понимание архитектурных компонентов этих систем поможет вам не только создать еще одну замечательную производную систему NoSQL, но также более ответственно использовать существующие версии систем.

13.7. Благодарности

Я признателен Jackie Carter, Mihir Kedia и анонимным рецензентам за их комментарии и пожелания по улучшению этой главы. Написание этой главы также было бы невозможным без многих лет самоотверженного труда сообщества NoSQL. Продолжайте созидательный труд!

Сноски

1. <http://hbase.apache.org/>
2. <http://project-voldemort.com/>
3. <http://cassandra.apache.org/>
4. <http://code.google.com/p/protobuf/>
5. <http://thrift.apache.org/>
6. <http://avro.apache.org/>
7. <http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html>
8. <http://redis.io/>
9. <http://couchdb.apache.org/>
10. <http://www.mongodb.org/>
11. http://www.basho.com/products_riak_overview.php
12. <http://www.hypergraphdb.org/index>
13. <http://neo4j.org/>
14. <http://memcached.org/>
15. <http://hadoop.apache.org/hdfs/>
16. <http://github.com/twitter/gizzard>
17. <http://hadoop.apache.org/zookeeper/>

14. Архитектура системы управления пакетами в Python

14.1 Введение

При разговоре о системах установки приложений обычно упоминают о двух подходах. Первый подход, характерный для Windows и Mac OS X, заключается в распространении самодостаточных пакетов приложений, процесс установки которых не должен зависеть от внешних факторов. Эта философия упрощает процесс управления приложениями: каждое приложение имеет свое отдельное "окружение" и его установка или удаление не влияет на другие части ОС. Если приложению для работы требуется нестандартная библиотека, эта библиотека включается в состав пакета для распространения приложения.

Второй подход, характерный для систем на основе ядра Linux, рассматривает программное обеспечение как набор небольших программных компонентов, называемых пакетами. Библиотеки добавляются в пакеты, причем любой пакет с библиотекой может зависеть от других пакетов. Процесс установки приложения может включать в себя процесс поиска и установки определенных версий множества других библиотек. Эти зависимости обычно доставляются из стандартного репозитория, содержащего тысячи пакетов. Данная философия обуславливает использование в дистрибутивах Linux таких сложных систем управления пакетами, как `dpkg` и `RPM` для отслеживания зависимостей и предотвращения установки двух приложений, использующих несовместимые версии одной и той же библиотеки.

У каждого подхода есть свои достоинства и недостатки. При использовании модульной системы, в которой каждый программный компонент может быть обновлен или заменен, снижаются затраты труда на обслуживание системы, так как каждая библиотека присутствует в единственном экземпляре и все приложения, использующие эту библиотеку, после обновления начнут использовать ее новую версию. Например, обновление безопасности системной библиотеки позволит одновременно защитить все приложения, использующие данную библиотеку, в отличие от случая, когда приложение использует отдельную копию библиотеки и установка обновления безопасности стано-

вится сложнее, особенно в том случае, когда различные приложения используют различные версии библиотеки.

Но модульность системы воспринимается как препятствие некоторыми разработчиками, так как они не могут контролировать работу своих приложений и их зависимостей. Для них было бы проще создать отдельный программный пакет для того, чтобы быть уверенными в том, что приложение будет работать в стабильном программном окружении, не попадая в "ад зависимостей" во время обновления системы.

Самодостаточные пакеты приложений также упрощают деятельность разработчика в том случае, когда он желает поддерживать несколько операционных систем. Некоторые проекты выпускаются в виде переносимых приложений, в которых исключено любое взаимодействие с файлами операционной системы, а работа осуществляется только с директорией приложения, даже в случае сохранения файлов журналов.

Система управления пакетами в Python разрабатывалась с использованием второго подхода - использовалось множество зависимостей для каждого пакета, а также система должна была быть так дружелюбна к разработчику, администратору и пользователю, как это возможно. К сожалению, она имела (и имеет) различные дефекты, обуславливающие и приводящие к разного рода проблемам: использованию неинтуитивных схем записи версий, наличию необрабатываемых файлов с данными, сложностям с повторной упаковкой и другим. Три года назад я и группа разработчиков Python решили повторно разработать эту систему для устранения вышеописанных проблем. Мы называем нашу группу Товариществом Разработчиков Системы Пакетов, а в данной главе будут описаны недостатки, которые мы пытались исправить, а также предложенные нами решения.

Терминология

При разговоре о пакете в Python имеется в виду директория с файлами исходного кода Python. Файлы исходного кода Python называются модулями. Такое описание приводит к непониманию при использовании слова "пакет", так как пакетом в ряде систем также называют релиз проекта.

Разработчики Python сами иногда путаются при разговоре об этом. Единственным путем преодоления этой двусмыслиности является использование термина "пакет Python" при разговоре о директории, содержащей модули Python. Термин "релиз" используется для описания одной версии проекта, а термин "дистрибутив" - для бинарных файлов или исходных кодов релиза, например, в форме файла архива tar или zip.

14.2. Трудности разработчиков Python

Большинство разработчиков Python желают, чтобы их программы были работоспособны в любом программном окружении. Также обычно они предпочитают использовать комбинацию стандартных библиотек языка Python и системно-зависимых библиотек. Но пока вы не создаете отдельные пакеты для каждой из систем управления пакетами, вам придется создавать релизы для Python - эти релизы создаются для установки с помощью соответствующей системы из состава Python независимо от используемой операционной системы и в теории должны соответствовать следующим требованиям:

- лица, поддерживающие пакеты для каждой операционной системы должны иметь возможность повторной упаковки каждого программного компонента,
- пакеты для программных компонентов, от которых зависит ваш пакет, также должны быть повторно упакованы в каждой операционной системе, а также
- системные зависимости должны быть четко описаны.

Иногда выполнение этих условий попросту невозможно. Например, Plone (полнофункциональная система управления содержимым сайта на основе Python) использует сотни небольших библиотек Python, которые не всегда доступны в виде пакетов в существующих системах управления пакетами. Это означает, что система Plone должна быть самодостаточным приложением, в комплект поставки которого включаются все необходимые программные компоненты. Для этого используется система сборки `zc.buildout`, с помощью которой происходит копирование всех используемых

программных компонентов и создание самодостаточного приложения, которое может работать на любой системе в рамках одной директории. На самом деле в итоге получается бинарный релиз, так как любой фрагмент кода на языке C может быть скомпилирован при сборке.

Это является большим подарком для разработчиков: им просто требуется описать зависимости с учетом приведенных ниже стандартов Python и использовать систему сборки `zc.buildout` для создания релиза приложения. Но, как обсуждалось ранее, данный тип релиза является отделенным от системы, поэтому не устроит большинство администраторов систем на основе Linux. Администраторы Windows-систем не будут против такой системы, но администраторы систем CentOS или Debian будут, так как эти системы выстраивают механизм управления пакетами исходя из предположения о том, что любой файл в системе является зарегистрированным, классифицированным и известным для инструментов администрирования.

Эти администраторы также захотят повторно упаковать ваше приложение в соответствии с используемыми ими стандартами. Нам необходимо ответить на вопрос: "Возможна ли система управления пакетами для Python, с помощью которой пакеты могли бы быть автоматически преобразованы в другие форматы?" В случае существования такой системы приложение или библиотека может быть установлено в любую операционную систему без необходимости повторной упаковки. В данном случае слово "автоматически" не обязательно означает, что вся работа по преобразованию должна вестись в рамках сценария: люди, работающие с системами управления пакетами `rpm` или `dpkg` скажут вам о том, что это невозможно - им всегда приходится добавлять специфические директивы при повторной упаковке проектов. Также они скажут вам о том, что они обычно испытывают сложности при повторной упаковке кода по той причине, что разработчики данного кода не учли нескольких основополагающих правил, применяемых при его упаковке.

Простой пример того, как вы можете создать трудности для людей, повторно упаковывающих ваш программный компонент, используя существующую систему управления пакетами Python: можно выпустить библиотеку с названием "MathUtils" версии "Fumanchu". Замечательному математику, разработавшему данную библиотеку, показалось забавным использовать имена своих котов в качестве версий проекта. Но как человек, сопровождающий пакет, узнает о том, что "Fumanchu" является именем второго кота, а первого кота зовут "Phil", поэтому версия "Fumanchu" вышла позднее версии "Phil"?

Может показаться странным, но такая ситуация вполне вероятна при использовании современных инструментов и стандартов. Плохо еще и то, что такие инструменты, как `easy_install` и `pip` используют нестандартные системы для отслеживания установленных файлов и сортируют версии "Fumanchu" и "Phil" в алфавитном порядке.

Другой проблемой является метод работы с файлами данных. Например, что будет, если ваше приложение использует базу данных SQLite? Если вы поместите базу данных в директорию вашего пакета, ваше приложение может работать некорректно, так как операционная система запрещает осуществлять запись в файлы, расположенные в данной части дерева файловой системы. Данный подход также нарушает предположения разработчиков операционных систем на основе ядра Linux о том, где должны находиться резервные копии данных приложений (в директории `/var`).

В реальности администраторы систем должны иметь возможность разместить файлы вашего приложения там, где они хотят, без нарушения работоспособности приложения, а вам необходимо довести до их сведения информацию об этих файлах. Поэтому давайте изменим формулировку вопроса: "Возможно ли создать такую систему управления пакетами в Python, которая бы предоставила всю необходимую информацию для повторной упаковки приложения с использованием существующей сторонней системы управления пакетами без необходимости чтения кода и устраивала бы всех?"

14.3 Современная архитектура системы управления пакетами

Пакет `Distutils`, поставляемый в составе стандартной библиотеки Python подвержен описанным выше проблемам. Так как он является стандартным пакетом, разработчики либо используют его, мирясь с его дефектами, либо используют такие более сложные инструменты, как `Setuptools`, использующий `Distutils` в качестве основы и предостав员ющий дополнительные функции, или `Distribute`, являющийся форком `Setuptools`. Также существует `Pip`, являющийся более сложным инструментом, созданным на основе `Setuptools`.

Однако, все эти новые инструменты основываются на `Distutils` и наследуют его недостатки. Попытки исправления недостатков `Distutils` предпринимались ранее, но его код настолько тесно связан с кодом других инструментов, что любое изменение даже внутренних интерфейсов вело к потенциальной регрессии всей экосистемы управления пакетами в Python.

Исходя из этого, мы решили прекратить работу над `Distutils` и начать разработку `Distutils2` на основе той же кодовой базы, не особо беспокоясь об обратной совместимости. Для понимания того, что и как было изменено, давайте рассмотрим подробнее архитектуру пакета `Distutils`.

14.3.1. Краткое описание и дефекты архитектуры `Distutils`

Пакет `Distutils` поддерживает команды, каждая из которых реализована в виде класса с методом `run`, который может быть вызван с различными параметрами. В `Distutils` также реализован класс `Distribution`, содержащий глобальные значения, к которым может обратиться любой класс с реализацией команды.

Для использования пакета `Distutils` разработчик добавляет в проект единственный модуль Python, традиционно называемый `setup.py`. В данном модуле реализуется обращение к главной точке входа в `Distutils`: функции `setup`. Эта функция принимает множество параметров, которые впоследствии хранятся с помощью экземпляра класса `Distribution` и используются классами команд. Ниже приведен пример передачи нескольких стандартных параметров, таких, как название и версия проекта, а также списка модулей проекта:

```
from distutils.core import setup

setup(name='MyProject', version='1.0', py_modules=['mycode.py'])
```

Впоследствии данный модуль может использоваться для выполнения таких команд `Distutils`, как `sdist`, которая создает архив для распространения исходного кода и размещает его в директории `dist`:

```
$ python setup.py sdist
```

Используя тот же сценарий, вы можете произвести установку проекта с помощью команды `install`:

```
$ python setup.py install
```

Пакет `Distutils` исполняет такие дополнительные команды, как:

- `upload` для загрузки архива с кодом для распространения в сетевой репозиторий,

- `register` для регистрации метаданных проекта в сетевом репозитории без обязательной загрузки архива с кодом для распространения,
- `bdist` для создания бинарного пакета для распространения, а также
- `bdist_msi` для создания установочного пакета с расширением `.msi` для Windows.

Также возможно получение дополнительной информации о проекте с помощью других параметров командной строки.

Таким образом, установка проекта или получение информации о нем всегда осуществляется путем использования пакета `Distutils` с помощью данного сценария. Например, для получения названия проекта можно использовать следующую команду:

```
$ python setup.py --name
MyProject
```

Следовательно, сценарий `setup.py` является инструментом для взаимодействия с проектом, независимо от того, нужно ли собрать, опубликовать или установить пакет. Разработчик описывает содержимое своего проекта с помощью параметров, передаваемых функции, а также использует этот сценарий для выполнения всех задач по работе с пакетом. Данный сценарий также используется для установки проекта в целевую систему.

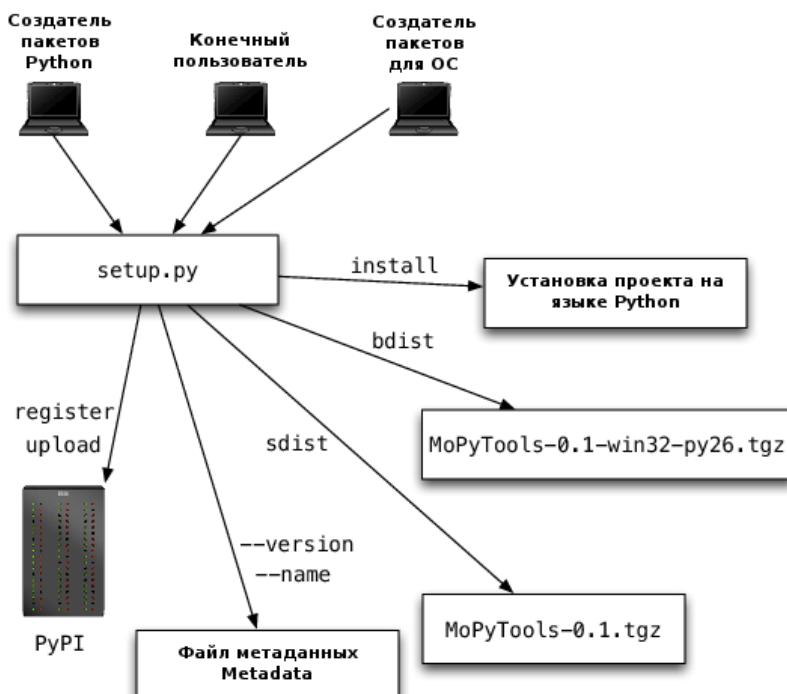


Рисунок 14.1: Функции модуля `setup.py`

Использование одного и того же модуля Python для упаковки, распространения и установки проекта является одним из главных дефектов архитектуры пакета `Distutils`. Например, если вы хотите получить название проекта `lxml`, сценарий `setup.py` выполнит множество действий помимо ожидаемого вывода строки:

```
$ python setup.py --name
Building lxml version 2.2.
NOTE: Trying to build without Cython, pre-generated 'src/lxml/lxml.etree.c'
needs to be available.
Using build configuration of libxslt 1.1.26
Building against libxml2/libxslt in the following directory: /usr/lib/lxml
```

Данная команда может даже не работать в некоторых проектах, так как их разработчики предполагают, что сценарий `setup.py` используется только для установки, поэтому остальные функции пакета `Distutils` будут использоваться ими только в период разработки. Множество вариантов использования сценария `setup.py` может просто привести в замешательство.

14.3 Современная архитектура системы управления пакетами

14.3.2. Метаданные и PyPI

Во время создания архива с исходным кодом для распространения пакет `Distutils` создает файл `Metadata`, соответствующий стандарту PEP 314¹. Он содержит статическую копию метаданных, представленных такими полями, как название проекта или версия релиза. Основные поля файла метаданных:

- `Name`: Название проекта.
- `Version`: Версия релиза.
- `Summary`: Краткое описание проекта в одной строке.
- `Description`: Подробное описание проекта.
- `Home-Page`: Стока URL для доступа к домашней странице проекта.
- `Author`: Имя автора.
- `Classifiers`: Классификаторы для проекта. Python использует список классификаторов для указания на лицензию, готовность релиза (`alpha`, `beta`, `final`) и другие параметры.
- `Requires`, `Provides` и `Obsoletes`: Используются для указания зависимостей от модулей.

Эти поля достаточно просто поставить в соответствие эквивалентным полям, используемым в других системах управления пакетами.

Каталог пакетов Python (The Python Package Index - PyPI)² является аналогичным CPAN центральным репозиторием пакетов, позволяющим регистрировать проекты и публиковать релизы с помощью команд пакета `Distutils register` и `upload`. С помощью команды `register` создается файл `Metadata`, который отправляется в PyPI, позволяя людям и таким используемым ими инструментам, как установщики, просматривать параметры проекта с помощью веб-страниц или веб-сервисов.

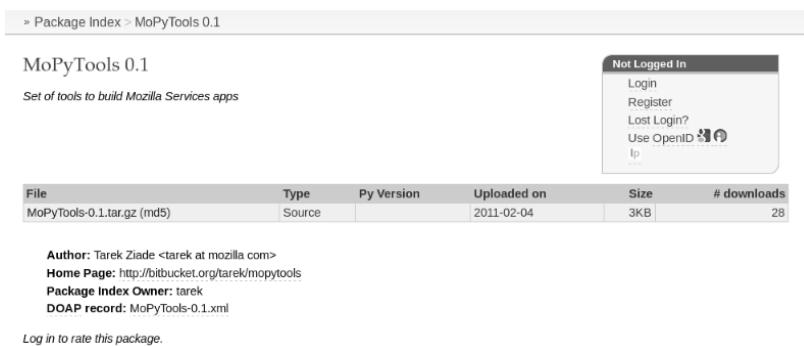


Рисунок 14.2: Репозиторий PyPI

Вы можете просмотреть проекты, выбрав классификаторы из поля `Classifiers` и получить имя автора и строку URL для доступа к домашней странице проекта. Между тем, поле `Requires` может быть использовано для указания зависимостей проекта от модулей Python. Параметр `requires` может быть использован для добавления модуля в поле `Requires` метаданных проекта:

```
from distutils.core import setup
```

```
setup(name='foo', version='1.0', requires=['ldap'])
```

Объявление зависимости от модуля `ldap` носит исключительно декларативный характер: никакой из инструментов или обработчиков не будет проверять наличие данного модуля. Данное решение было бы удачным в том случае, если бы Python использовал зависимости на уровне модулей, задаваемые с помощью ключевого слова `require` аналогично Perl. После этого задача сценария установки заключалась бы в поиске зависимостей в PyPI и установке их; таким образом функционирует CPAN. Но это не возможно в Python, так как модуль с именем `ldap` может использоваться в любом проекте Python. Так как пакет `Distutils` позволяет публиковать проекты, которые могут содержать по нескольку пакетов и модулей, данное поле метаданных полностью бесполезно.

Другим дефектом файлов `Metadata` является тот факт, что они создаются с помощью сценария `Python`, следовательно, являются специфичными для платформы, на которой происходило создание. Например, проект, предоставляющий специфичные для Windows возможности, может использовать следующий файл `setup.py`:

```
from distutils.core import setup

setup(name='foo', version='1.0', requires=['win32com'])
```

Но этот файл предполагает работу проекта только под управлением Windows даже в том случае, когда предоставляются переносимые функции. Одним способом решения этой проблемы является использование специфичного для Windows параметра `requires`.

```
from distutils.core import setup
import sys

if sys.platform == 'win32':
    setup(name='foo', version='1.0', requires=['win32com'])
else:
    setup(name='foo', version='1.0')
```

Это объявление частично решает проблему. Вспомните о том, что сценарий используется для создания архивов с исходным кодом, которые впоследствии публикуются с помощью PyPI. Это значит, что статический файл `Metadata`, отправляющийся для публикации в PyPI, зависит от платформы, на которой происходит его формирование. Другими словами, нет способа для указания в поле метаданных того, что пакет зависит от платформы.

14.3 Современная архитектура системы управления пакетами

14.3.3. Архитектура PyPI

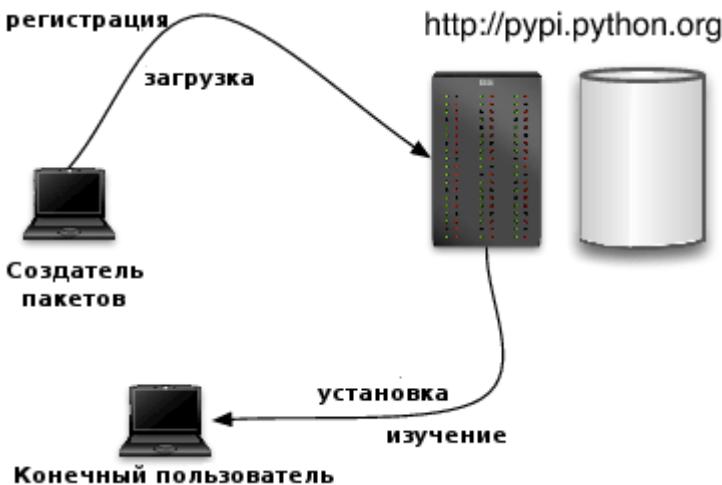


Рисунок 14.3: Процесс работы PyPI

Как было сказано ранее, PyPI является центральным каталогом проектов на языке Python, с помощью которого люди могут просматривать существующие разделенные на категории проекты или зарегистрировать свои работы. Архивы с исходным кодом и бинарными файлами для распространения могут быть загружены и добавлены к существующему проекту, после чего становится возможным их скачивание для установки или изучения. PyPI также предоставляет веб-сервисы, которые могут использоваться такими инструментами, как установщики.

Регистрация проектов и загрузка архивов

Регистрация проекта в каталоге PyPI осуществляется с помощью команды `register` пакета `Distutils`. С помощью этой команды формируется POST-запрос, содержащий метаданные проекта вне зависимости от его версии. Запрос содержит заголовок авторизации, так как PyPI использует механизм аутентификации Basic для того, чтобы каждый зарегистрированный проект был ассоциирован с пользователем, ранее зарегистрировавшимся в PyPI. Аутентификационные данные хранятся в локальном файле настроек `Distutils` или вводятся после запроса при каждом вызове команды `register`. Пример использования этой команды:

```
$ python setup.py register
running register
Registering MPTools to http://pypi.python.org/pypi
Server response (200): OK
```

Каждый зарегистрированный проект получает веб-страницу, содержащую HTML-версию метаданных, и создатели пакетов могут загружать архивы для распространения в PyPI с помощью команды `upload`:

```
$ python setup.py sdist upload
running sdist
...
running upload
Submitting dist/mopytools-0.1.tar.gz to http://pypi.python.org/pypi
Server response (200): OK
```

Также возможно перенаправление пользователей на другой адрес с помощью поля метаданных `Download-URL` вместо загрузки файлов напрямую на PyPI.

Запросы к PyPI

Помимо HTML-страниц, генерируемых PyPI для пользователей, с помощью инструментов могут быть использованы две службы: простой протокол каталога и API XML-RPC.

Простой протокол каталога используется при доступе к адресу <http://pypi.python.org/simple/>, причем в ответ отправляется обычная HTML-страница, содержащая относительные ссылки на страницы каждого из зарегистрированных проектов:

```
<html><head><title>Simple Index</title></head><body>
:
:
<a href='MontyLingua/'>MontyLingua</a><br/>
<a href='mootiro_web/'>mootiro_web</a><br/>
<a href='Mopidy/'>Mopidy</a><br/>
<a href='mopowg/'>mopowg</a><br/>
<a href='MOPPY/'>MOPPY</a><br/>
<a href='MPTools/'>MPTools</a><br/>
<a href='morbid/'>morbid</a><br/>
<a href='Morelia/'>Morelia</a><br/>
<a href='morse/'>morse</a><br/>
:
:
</body></html>
```

Например, для проекта MPTools используется ссылка `MPTools/`, что означает наличие проекта в каталоге. Страница, на которую ведет ссылка, содержит ссылки на все относящиеся к проекту ресурсы:

- ссылки на все архивы с кодом для распространения, хранящиеся на PyPI
- ссылки на все домашние страницы проекта, заданные в файле `Metadata`, для каждой зарегистрированной версии проекта
- ссылки на файлы, заданные в файле `Metadata`, также для каждой версии проекта.

Страница проекта MPTools содержит следующие ссылки:

```
<html><head><title>Links for MPTools</title></head>
<body><h1>Links for MPTools</h1>
<a href="../../packages/source/M/MPTools/MPTools-0.1.tar.gz">MPTools-0.1.tar.gz</a><br/>
<a href="http://bitbucket.org/tarek/mopytools" rel="homepage">0.1 home_page</a><br/>
</body></html>
```

Такие инструменты, как установщики, при необходимости проверки наличия проекта могут поискать его в каталоге или просто проверить, существует ли страница с адресом http://pypi.python.org/simple/НАЗВАНИЕ_ПРОЕКТА/.

Этот протокол имеет два недостатка. Во-первых, на данный момент каталог PyPI работает на одном сервере, и, хотя разработчики обычно имеют локальные копии его содержимого, мы сталкивались с несколькими случаями недоступности сервера за последние два года, что приводило к остановке работы с установщиками, использующими PyPI для получения списка зависимостей, необходимого для сборки проекта. Например, при сборке приложения Plone генерируется несколько сотен запросов к каталогу PyPI для получения всех необходимых данных, поэтому каталог PyPI в некоторых случаях может оказаться единой точкой отказа.

Во-вторых, в тех случаях, когда архивы для распространения не хранятся на сервере PyPI и на странице проекта приводится ссылка для их скачивания, установщикам придется переходить по ссылке и рассчитывать на работоспособность стороннего сервера и наличие необходимого архива на этом сервере. Эти неопределенности снижают надежность функционирования процесса сборки при работе с простым протоколом каталога.

Целью простого протокола каталога является передача установщикам списка ссылок, необходимых им для установки проекта. С помощью этого протокола не передаются метаданные проекта;

напротив, существуют методы XML-RPC для получения дополнительной информации о зарегистрированных проектах.

```
<<< import xmlrpclib
<<< import pprint
<<< client = xmlrpclib.ServerProxy('http://pypi.python.org/pypi')
<<< client.package_releases('MPTools')
['0.1']
<<< pprint.pprint(client.release_urls('MPTools', '0.1'))
[{'comment_text': '',
'downloads': 28,
'filename': 'MPTools-0.1.tar.gz',
'has_sig': False,
'md5_digest': '6b06752d62c4bfffefb65cd5c9b7111a',
'packagetype': 'sdist',
'python_version': 'source',
'size': 3684,
'upload_time': '',
'url': 'http://pypi.python.org/packages/source/M/MPTools/MPTools-0.1.tar.gz'}]
>>> pprint.pprint(client.release_data('MPTools', '0.1'))
{'author': 'Tarek Ziade',
'author_email': 'tarek@mozilla.com',
'classifiers': [],
'description': 'UNKNOWN',
'download_url': 'UNKNOWN',
'home_page': 'http://bitbucket.org/tarek/mopytools',
'keywords': None,
'license': 'UNKNOWN',
'maintainer': None,
'maintainer_email': None,
'name': 'MPTools',
'package_url': 'http://pypi.python.org/pypi/MPTools',
'platform': 'UNKNOWN',
'release_url': 'http://pypi.python.org/pypi/MPTools/0.1',
'requires_python': None,
'stable_version': None,
'summary': 'Set of tools to build Mozilla Services apps',
'version': '0.1'}
```

Недостатком этого подхода является тот факт, что часть передаваемой посредством API XML-RPC информации следовало бы хранить в статических файлах и предоставлять на странице проекта по простому протоколу каталога для упрощения работы клиентских инструментов. Это решение позволило бы также сократить объем дополнительной работы, выполняемой PyPI для обработки этих запросов. Неплохо иметь доступ к динамически изменяющимся данным, таким, как количество загрузок архива для распространения для их публикации на специализированном веб-сервисе, но не имеет смысла использовать две различные службы для получения статических данных, относящихся к проекту.

14.3 Современная архитектура системы управления пакетами

14.3.4. Архитектура системы установки Python

Если вы устанавливаете проект на языке Python с помощью команды `python setup.py install`, пакет `Distutils`, включенный в стандартную библиотеку, скопирует файлы в вашу систему.

- Модули и пакеты Python будут размещены в директории `Python`, используемой при запуске интерпретатора: в последней версии Ubuntu в директории `/usr/local/lib/python2.6/dist-packages/` и в Fedora в директории `/usr/local/lib/python2.6/sites-packages/`.
- Файлы данных проекта могут быть размещены в любой системной директории.

- Исполняемый сценарий будет размещен в системной директории `bin`. В зависимости от платформы может использоваться директория `/usr/local/bin` или другая директория, заданная при установке Python.

С версии Python 2.5 файлы метаданных копируются вместе с модулями и пакетами в файлы с именами проект-версия.egg-info. Например, для проекта `virtualenv` будет использоваться имя файла `virtualenv-1.4.9.egg-info`. Эти файлы могут использоваться в качестве базы данных установленных проектов, так как в ходе их обхода становится возможным составление списка проектов вместе с их версиями. Однако, установщик из пакета `Distutils` не сохраняет список файлов, установленных в систему. Другими словами, нет способа удаления всех файлов, скопированных в систему. Это досадно, так как команда `install` поддерживает параметр `--record`, который может использоваться для записи списка установленных файлов в текстовый файл. Как бы то ни было, этот параметр не используется по умолчанию и в документации пакета `Distutils` содержится лишь краткое упоминание о нем.

14.3 Современная архитектура системы управления пакетами

14.3.5. `Setuptools`, `Pip` и аналогичные проекты

Как упоминалось во введении, в рамках некоторых проектов предпринимались попытки исправления определенных недоработок пакета `Distutils` с переменным успехом.

Вопрос зависимостей

Каталог PyPI позволяет разработчикам публиковать проекты на языке Python, содержащие несколько модулей для организации пакетов Python. Но в то же время проекты должны объявлять зависимости на уровне модулей с помощью директивы `Require`. Оба подхода разумны, но вот их комбинация - нет.

Правильным решением было бы объявление зависимостей на уровне проекта, что и было сделано в проекте `Setuptools`, добавившим эту функцию к `Distutils`. Он также предоставлял сценарий `easy_install` для автоматического получения и установки зависимостей путем поиска в каталоге PyPI. На практике зависимости уровня модулей никогда не использовались в полной мере, а вместо них использовались расширения `Setuptools`. Но так как эти параметры были специфичными для `Setuptools` и игнорировались `Distutils` или PyPI, в рамках пакета `Setuptools` был создан собственный стандарт, ставший решением, исправляющим недоработку в архитектуре системы управления пакетами.

Сценарию `easy_install` приходится скачивать архив проекта и запускать сценарий `setup.py` из его состава для получения необходимых метаданных, а также ему приходится повторять этот процесс для каждой зависимости. Граф зависимостей строится последовательно после каждого скачивания.

Даже если метаданные приняты в каталог PyPI и доступны в сети, сценарию `easy_install` также будет необходимо скачивать все архивы, ведь, как говорилось ранее, публикуемые с помощью PyPI метаданные зависят от платформы, которая использовалась для их загрузки и может отличаться от целевой платформы. Тем не менее, такая возможность установки проекта вместе с его зависимостями была достаточна в 90% случаев и была замечательной функцией. Поэтому пакет `Setuptools` получил широкое распространение, хотя он и был подвержен другим недостаткам:

- Если установка зависимости проходила неудачно, не было возможности отката изменений, поэтому после установки проект мог остаться в нерабочем состоянии.
- Граф зависимостей строился в ходе установки пакетов, поэтому в случае обнаружения конфликта в ходе установки проект мог также остаться в нерабочем состоянии.

Вопрос удаления файлов

Пакет `Setuptools` не предоставляет инструмента для удаления файлов, хотя его специфические метаданные и могут содержать список установленных файлов. Проект `Pip`, с другой стороны, расширил метаданные `Setuptools` для записи списка установленных файлов, и, таким образом добавил возможность удаления файлов. Однако, в данном случае использовался еще один набор специфических метаданных, поэтому отдельная установка `Python` должна была содержать:

- Метаданные `egg-info` от `Distutils` в форме одного файла метаданных.
- Метаданные `egg-info` от `Setuptools` в форме директории с файлами метаданных с специфическими для `Setuptools` параметрами.
- Метаданные `egg-info` от `Pip` в форме расширения предыдущих метаданных.
- Любые метаданные, создаваемые системами управления пакетами системы.

14.3 Современная архитектура системы управления пакетами

14.3.6. Как насчет файлов данных?

В `Distutils` файлы данных могут быть установлены в любую директорию. Если вы зададите список файлов данных пакета в сценарии `setup.py` подобным образом:

```
setup(...,
    packages=['mypkg'],
    package_dir={'mypkg': 'src/mypkg'},
    package_data={'mypkg': ['data/*.dat']},
)
```

то все файлы с расширением `.dat` в проекте `mypkg` будут включены в архив для распространения и в конечном счете установлены вместе с модулями `Python` с помощью системы установки.

Для файлов, устанавливаемых вне директории проекта `Python`, существует другой параметр, позволяющий хранить файлы в архиве, но размещать их в заданных директориях при установке:

```
setup(...,
    data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                ('config', ['cfg/data.cfg']),
                ('/etc/init.d', ['init-script'])]
)
```

Это очень плохие новости для поддерживающих пакеты для операционных систем лиц по нескольким причинам:

- Файлы данных не являются частью метаданных, поэтому работающим над пакетами лицам придется исследовать файл `setup.py`, а иногда и код проекта.
- Разработчик не должен принимать единоличное решение о том, где будут расположены файлы данных в целевой системе.
- Для файлов данных не существует категорий: изображения, страницы руководств `man` и все другие файлы обрабатываются аналогичным образом.

Человек, желающий повторно упаковать проект с такими файлами, не имеет иного пути, кроме как разработать патч для файла `setup.py` для того, чтобы он работал так, как требуется на данной платформе. Для этого ему придется исследовать код и заменить каждую строку, где используются эти файлы, так как разработчик задал их расположение в системе. Проекты `Setuptools` и `Pip` не улучшили эту ситуацию.

14.4. Усовершенствованные стандарты

Мы закончили рассмотрение запутанного и непоследовательного окружения для управления пакетами, в котором все действия выполняются с помощью единственного модуля Python с использованием неполной формы метаданных и отсутствием возможности для описания всех файлов проекта. Теперь поговорим о том, как улучшить ситуацию.

14.4.1. Метаданные

Первым шагом является усовершенствование стандарта метаданных. Стандарт PEP 345 описывает новую версию формата метаданных, которая предусматривает:

- более корректный метод указания версий
- зависимости уровня проекта
- стандартный метод для описания зависимых от платформы значений

Версия

Одной из целей изменения стандарта метаданных является предоставление возможности всем работающим с проектами Python инструментам классифицировать их аналогичным образом. В случае версий это означает, что каждый инструмент должен быть в состоянии установить то, что версия "1.1" была выпущена после версии "1.0". Но если используются специфические схемы указания версий, эта задача значительно усложняется.

Единственной возможностью установления постоянного формата версий является публикация стандарта, которому будут следовать проекты. Выбранная нами схема является классической схемой на основе последовательности версий. Как описано в стандарте PEP 386, для указания версий используется следующий формат строки:

N.N[.N]+[{a|b|c|rc}N[.N]+] [.postN] [.devN]

где:

- N является целочисленным значением. Вы можете использовать столько значений N, сколько хотите, при этом следует учитывать, что значения должны быть разделены точками и их должно быть не менее двух (ВЕРСИЯ.ПОДВЕРСИЯ).
- a,b,c и rc являются обозначениями alpha-версии, beta-версии и кандидата в релизы. После них следует целочисленное значение. Для кандидатов в релизы используются два последних обозначения, так как мы хотели сделать схему совместимой со схемой Python, в которой используется rc. При этом нам кажется, что использовать с проще.
- dev с последующим целочисленным значением является обозначением версии для разработчиков.
- post с последующим целочисленным значением является обозначением версии, выпущенной после релиза.

В зависимости от процесса разработки, обозначения dev и post могут использоваться для всех промежуточных версий, выпускаемых между двумя финальными релизами. Для большей части релизов в процессе разработки используется обозначение dev.

Используя эту схему, стандарт PEP 386 определяет строгую последовательность:

- alpha-версия < beta-версия < кандидат в релизы < финальный релиз
- версия для разработчиков < обычная версия < версия, выпущенная после релиза, где обычной версией может быть alpha-версия, beta-версия, кандидат в релизы или финальная версия

Ниже приведен пример указания версий:

```
1.0a1 < 1.0a2.dev456 < 1.0a2 < 1.0a2.1.dev456
< 1.0a2.1 < 1.0b1.dev456 < 1.0b2 < 1.0b2.post345
```

```
< 1.0c1.dev456 < 1.0c1 < 1.0.dev456 < 1.0
< 1.0.post456.dev34 < 1.0.post456
```

Целью разработки этой схемы было простое преобразование версий пакетов Python в версии для других систем управления пакетами. В данный момент каталог PyPI отклоняет любые загружаемые проекты, использующие стандарт PEP 345 для метаданных с версией, не соответствующей стандарту PEP 386.

Зависимости

Стандарт PEP 345 описывает три новых поля, заменяющих поля из стандарта PEP 314 `Requires`, `Provides` и `Obsoletes`. Этими полями являются `Requires-Dist`, `Provides-Dist` и `Obsoletes-Dist`, которые могут использоваться по нескольку раз в метаданных.

Каждая строка поля `Requires-Dist` должна содержать название другого проекта в формате `Distutils`, требуемого для работы данного проекта. Формат строки идентичен формату названия проекта в `Distutils` (т.е. формату, используемому при объявлении названия проекта в поле `Name`), после которого в скобках может следовать объявление версии. Эти названия проектов в формате `Distutils` должны соответствовать названиям проектов, используемых в PyPI, а объявления версий - соответствовать стандарту PEP 386. Некоторые примеры приведены ниже:

```
Requires-Dist: pkginfo
Requires-Dist: PasteDeploy
Requires-Dist: zope.interface (>3.5.0)
```

Поле `Provides-Dist` используется для указания дополнительных названий проектов в рамках данного проекта. Оно полезно в том случае, когда несколько проектов объединяются. Например, проект ZODB может включать в свой состав проект `transaction` и использовать следующее объявление:

```
Provides-Dist: transaction
```

Поле `Obsoletes-Dist` полезно в том случае, когда необходимо указать на то, что версия другого проекта становится устаревшей после установки проекта:

```
Obsoletes-Dist: OldName
```

Маркеры окружения

Маркер окружения является условным переходом, зависящим от окружения исполнения и добавляемым в конец поля после точки с запятой. Некоторые примеры приведены ниже:

```
Requires-Dist: pywin32 (>1.0); sys.platform == 'win32'
Obsoletes-Dist: pywin31; sys.platform == 'win32'
Requires-Dist: foo (1,!>1.3); platform.machine == 'i386'
Requires-Dist: bar; python_version == '2.4' or python_version == '2.5'
Requires-External: libxslt; 'linux' in sys.platform
```

Микроязык для маркеров окружения сознательно разработан с учетом простоты понимания разработчиками, не знакомыми с языком Python: он сравнивает строки с помощью операторов `==` и `in` (и противоположных) и позволяет использовать обычные логические комбинации. Поля из стандарта PEP 345, совместно с которыми могут использоваться эти маркеры:

- Requires-Python
- Requires-External
- Requires-Dist
- Provides-Dist
- Obsoletes-Dist
- Classifier

14.4. Усовершенствованные стандарты

14.4.2. Что установлено?

Использование единого формата списков установленных файлов для всех инструментов Python необходимо для их взаимодействия. Если мы хотим, чтобы установщик A определял то, что установщик B установил ранее проект Foo, оба этих установщика должны использовать и обновлять одну и ту же базу данных установленных проектов.

Конечно же, в идеальном случае пользователи должны использовать один установщик в их системе, но им может понадобиться перейти к использованию нового установщика, предоставляющего специфические возможности. Например, в составе Mac OS X поставляется пакет `Setuptools`, поэтому пользователи автоматически получают в свое распоряжение сценарий `easy_install`. Если они пожелают перейти к использованию нового инструмента, им придется воспользоваться инструментом, который поддерживает обратную совместимость с предыдущим.

Другой проблемой является использование установщика Python на платформах, использующих системы управления пакетами, аналогичные RPM, так как в этом случае нет способа информировать систему об установке проекта. Вся сложность заключается в том, что хотя система управления пакетами Python и может каким-либо образом получить доступ к центральной системе управления пакетами, возникнет необходимость сопоставления метаданных Python с метаданными центральной системы управления пакетами. Название проекта, например, может быть различным для этих систем. Это может произойти по нескольким причинам. Наиболее известной причиной является конфликт названий: другой проект вне экосистемы Python может использовать то же название в рамках системы управления пакетами RPM. Другой причиной является использование префикса `python`, которое нарушает соглашение, используемое при создании пакетов для платформы. Например, если название вашего проекта `foo-python`, велика вероятность, что пакет RPM в системе Fedora будет назван `python-foo`.

Одним из способов преодоления данной проблемы является глобальная установка Python с помощью центральной системы управления пакетами и работа в изолированном окружении. Такие инструменты, как `Virtualenv` могут помочь в этом случае.

В любом случае нам необходим единый формат установки пакетов в Python, так как другим системам установки пакетов также необходимо взаимодействовать с системой установки пакетов Python в тех случаях, когда пакеты Python устанавливаются с помощью них. Как только сторонняя система управления пакетами регистрирует установленный проект в своей базе данных, ей необходимо сгенерировать корректные метаданные установки для Python таким образом, чтобы проекты определялись как установленные установщиками Python и другими API, запрашивающими данные об установленных пакетах в Python.

Сопоставление метаданных актуально и в данном случае: так как система управления пакетами RPM обладает информацией о том, какие проекты Python устанавливаются с помощью нее, она может генерировать корректные метаданные уровня Python. Например, данной системе известно, что названию `python26-webob` соответствует название `WebOb` в экосистеме PyPI.

Вернемся к нашему стандарту: стандарт PEP 376 описывает формат устанавливаемых пакетов, аналогичный используемым форматам проектов `Setuptools` и `Pip`. Описанная структура представлена директорией с расширением `dst-info`, которая содержит файлы:

- `METADATA`: метаданные, как описано в стандартах PEP 345, PEP 314 и PEP 241.
- `RECORD`: список установленных файлов в формате, аналогичном `csv`.
- `INSTALLER`: название инструмента, использованного для установки проекта.
- `REQUESTED`: наличие данного файла указывает на то, что установка проекта была произведена по явному запросу (т.е. пакет установлен не по зависимости).

Как только все существующие инструменты будут поддерживать этот формат, у нас будет возможность управлять проектами Python вне зависимости от определенного установщика и его возможностей. Также, поскольку стандарт PEP 376 описывает хранилище метаданных в виде директории, расширение метаданных сводится к добавлению новых файлов. На самом деле, новый файл метаданных с именем `RESOURCES`, описанный в следующем разделе, может быть добавлен в ближайшем будущем без внесения изменений в стандарт PEP 376. В конечном счете, если этот новый файл окажется полезным для всех инструментов, его описание будет добавлено в стандарт PEP.

14.4. Усовершенствованные стандарты

14.4.3. Архитектурные решения в отношении работы с файлами данных

Как описывалось ранее, нам необходимо предоставить возможность принятия решений о местах размещения файлов данных ответственным за создание пакетов лицам, при этом их действия не должны приводить к неработоспособности кода. В то же время разработчик должен иметь возможность работы с файлами данных, не беспокоясь о их размещении. Наше решение является обычным применением обходных путей.

Использование файлов данных

Предположим, что вашему приложению `MPTools` требуется использовать конфигурационный файл. Разработчик должен поместить этот файл в пакет Python и использовать переменную `__file__` для доступа к нему:

```
import os

here = os.path.dirname(__file__)
cfg = open(os.path.join(here, 'config', 'mopy.cfg'))
```

Подразумевается, что конфигурационные файлы устанавливаются аналогично файлам с кодом и разработчик обязан разместить эти файлы вместе с кодом: в этом примере в поддиректории с именем `config`.

Новые архитектурные решения в отношении работы с файлами данных позволяют использовать дерево директорий проекта в качестве корневой директории для всех файлов, а также позволяют получать доступ к любому файлу в этом дереве независимо от того, расположен ли он в пакете Python или в простой директории. Эти решения позволили разработчикам создавать отдельную директорию для файлов данных и получать доступ к ним с помощью метода `pkgutil.open`:

```
import os
import pkgutil

# Открыть файл, расположенный в директории config/mopy.cfg проекта MPTools
cfg = pkgutil.open('MPTools', 'config/mopy.cfg')
```

Метод `pkgutil.open` получает доступ к метаданным проекта и проверяет наличие файла `RESOURCES`. В этом файле находится простой список соответствия для имен файлов и возможных мест их расположения в системе:

```
config/mopy.cfg {confdir}/{distribution.name}
```

В данном случае переменная `{confdir}` указывает на системную директорию для файлов конфигурации, а переменная `{distribution.name}` содержит имя проекта Python, извлеченное из метаданных.

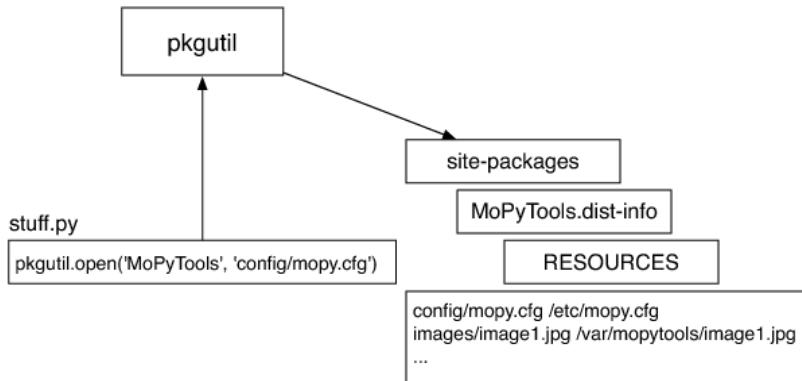


Рисунок 14.4: Поиск файла

Как только данный файл метаданных `RESOURCES` создается во время установки, у разработчика появляется возможность с помощью API узнать о расположении файла `mopy.cfg`. И так как путь `config/mopy.cfg` относится к дереву директорий проекта, появляется возможность для реализации режима разработчика, в котором метаданные для проекта генерируются в директории проекта и путь к ним добавляется в список директорий поиска `pkgutil`.

Объявление файлов данных

На практике проект может установить место расположения файлов данных, объявив сопоставление в файле `setup.cfg`. Сопоставление является списком кортежей формата (шаблон в формате `glob`, целевой путь). Каждый шаблон соответствует одному или нескольким файлам в дереве проекта, а целевой путь указывает путь для установки и может содержать переменные в фигурных скобках. Например, файл `setup.cfg` проекта `MPTools` может выглядеть подобным образом:

```
[files]
resources =
    config/mopy.cfg {confdir}/{application.name}/
    images/*.jpg     {datadir}/{application.name}/
```

В модуле `sysconfig` объявлен и документирован список специальных переменных, которые могут быть использованы, а также их значения по умолчанию для каждой из платформ. Например, значением переменной `{confdir}` является строка `/etc` в Linux. Таким образом, установщики могут использовать это сопоставление вместе с модулем `sysconfig` во время установки для получения путей для копирования файлов. В конечном счете, установщики будут генерировать упоминавшийся ранее файл `RESOURCES` в составе метаданных и, таким образом, расположение этих файлов сможет быть установлено впоследствии с помощью функций модуля `pkgutil`.

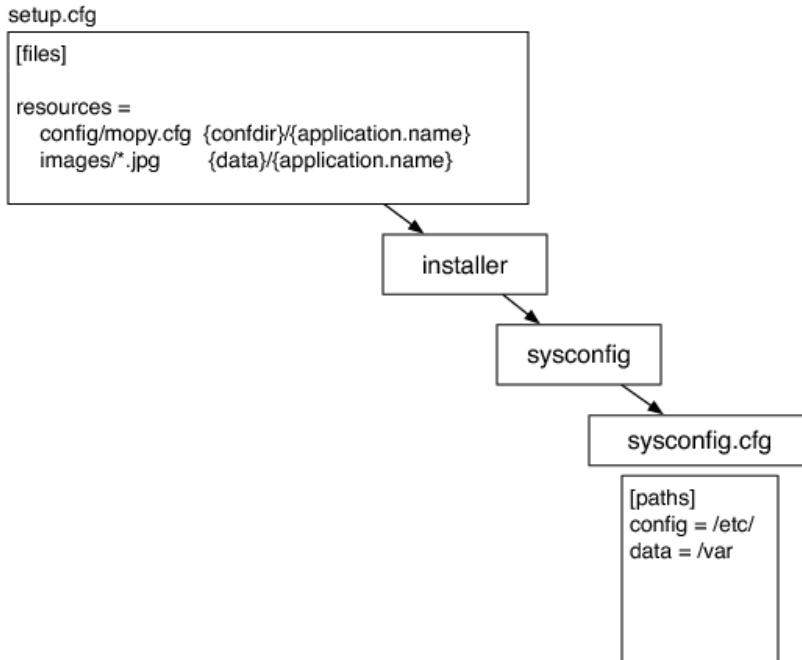


Рисунок 14.5: Установщик

14.4. Усовершенствованные стандарты

14.4.4. Усовершенствования каталога PyPI

Как я говорил ранее, каталог PyPI может оказаться единой точкой отказа. Стандарт PEP 380 направлен на устранение данной проблемы и описывает протокол зеркалирования, позволяющий пользователям обращаться к альтернативным серверам в случае неработоспособности центрального сервера PyPI. Целью данных усовершенствований является предоставление возможности членам сообщества вводить в строй зеркальные серверы по всему миру.

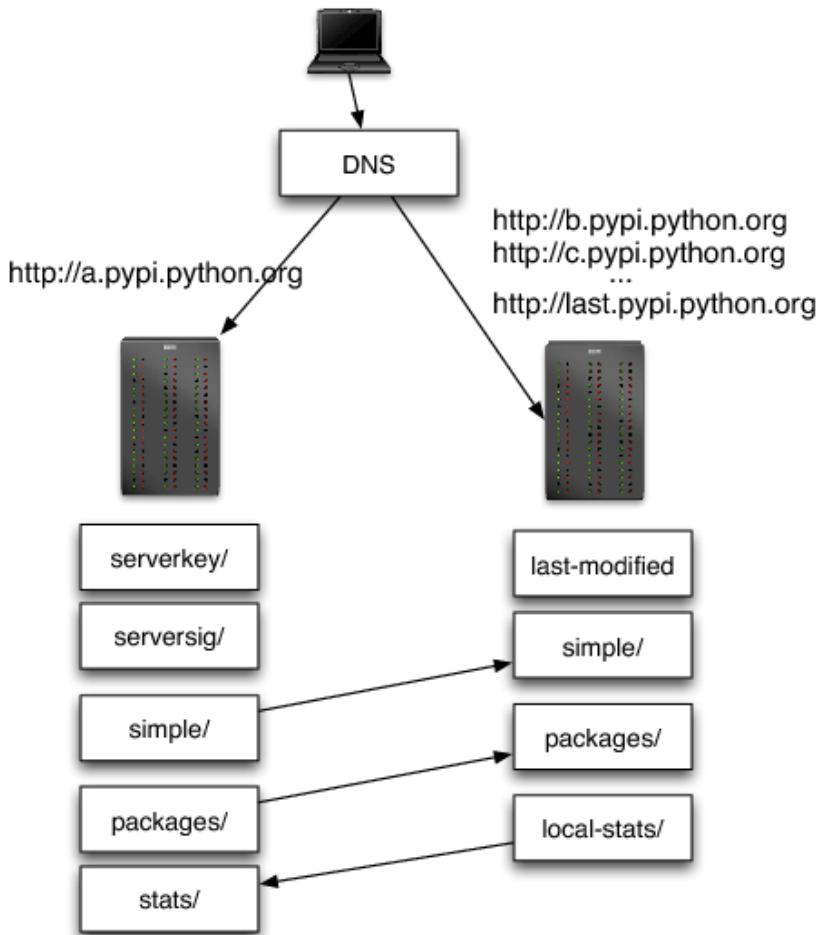


Рисунок 14.6: Зеркалирование

Список зеркал формируется в виде списка имен узлов в форме `x.pypi.python.org`, где `x` является последовательностью буквенных символов `a, b, c, ..., aa, ab, ...`. Сервер с именем `a.pypi.python.org` является мастер-сервером, а имена зеркальных серверов начинаются с символа `b`. Запись CNAME `last.pypi.python.org` указывает на имя последнего узла, поэтому клиенты, использующие PyPI, могут составить список зеркальных серверов, получив запись CNAME.

Например, данный вызов сообщает пользователю о том, что последним зеркальным сервером является сервер с именем `h.pypi.python.org`, а это значит, что каталог PyPI на данный момент использует 6 серверов зеркал (обозначенных символами от `b` до `h`):

```

>>> import socket
>>> socket.gethostbyname_ex('last.pypi.python.org')[0]
['h.pypi.python.org']
  
```

Теоретически данный протокол позволяет клиентам осуществлять переадресацию запросов на ближайший к ним зеркальный сервер, устанавливая расположение серверов на основе их IP-адресов, а также переходить к другому зеркальному серверу в случае неработоспособности зеркального сервера или мастер-сервера. Протокол зеркалирования сам по себе является более сложным, чем простой протокол rsync, так как мы хотели получать точную статистику скачиваний и поддерживать минимальные меры безопасности.

Синхронизация

Зеркальные сервера должны снижать объем данных, передаваемых между центральным и зеркальным сервером. Для достижения этой цели они должны использовать вызов `changelog` интерфейса XML-RPC PyPI и обновлять только те пакеты, содержимое которых было изменено после

последней проверки. Для каждого пакета Р зеркальными серверами должны быть скопированы документы из директорий `/simple/P/` и `/serversig/P/`.

Если пакет был удален на центральном сервере, на зеркальных серверах этот пакет также должен быть удален вместе со всеми ассоциированными файлами. Для определения модификации файлов пакетов зеркальные сервера могут кэшировать параметр ETag для каждого файла и запрашивать файл с использованием заголовка `If-None-Match`, не принимая сам файл. Как только синхронизация завершается, зеркальный сервер помещает текущую дату в файл `/last-modified`.

Распространение статистических данных

Когда вы скачиваете релиз с одного из зеркал, информацию о скачивании передается мастер-серверу PyPI и другим зеркальным серверам по соответствующему протоколу. Этот принцип позволяет быть уверенным в том, что люди или инструменты, просматривающие каталог PyPI в поисках количества скачиваний релиза, получат значение, просуммированное по всем зеркальным серверам.

Статистические данные обрабатываются и заносятся файлы формата CSV со статистикой за день и за неделю, расположенные в директории `stats` на центральном сервере PyPI. Каждый зеркальный сервер должен использовать директорию `local-stats` для хранения своей собственной статистики. Каждый файл содержит данные о количестве скачиваний для каждого архива, сгруппированные по используемым заголовкам "User-Agent". Центральный сервер ежедневно посещает зеркальные сервера для сбора их статистических данных и добавления их в файлы из глобальной директории `stats`, поэтому каждый зеркальный сервер должен обновлять содержимое директории `/local-stats` как минимум раз в день.

Подлинность зеркальных серверов

В распределенной системе зеркальных серверов клиентам может понадобиться выяснить, являются ли копии данных подлинными. Возможные опасности включают в себя:

- компрометацию центрального каталога
- подмену данных на зеркальных серверах
- атаку перехвата данных между центральным сервером и конечным пользователем

Для установления факта реализации первого типа атаки авторы пакетов должны подписывать свои пакеты с помощью ключей PGP, позволяя таким образом пользователям проверять, выпущен ли пакет автором, которому они доверяют. Протокол зеркалирования сам по себе предусматривает меры борьбы только со второй угрозой, хотя в нем и предпринята попытка определения факта реализации атак перехвата данных.

На центральном сервере в директории `/serverkey` хранится ключ DSA в формате PEM, сгенерированный с помощью команды `openssl dsa -pubout`³. Данная директория не должна зеркалироваться и клиенты должны получать официальный ключ напрямую из каталога PyPI или использовать его копию, предоставляемую в комплекте с клиентскими приложениями PyPI. Зеркальные сервера все же должны скачивать ключ для того, чтобы определять факт продления его срока действия.

Для каждого пакета зеркалируемая подпись находится в директории `/serversig/package`. Это подпись DSA для параллельной страницы с URL `/simple/package` в форме DER с использованием алгоритма SHA-1 с DSA⁴.

Клиенты, использующие зеркальные сервера, должны выполнить следующие действия для проверки пакета:

1. Скачать страницу `/simple` и рассчитать ее хэш с помощью алгоритма SHA-1.
2. Сформировать подпись DSA на основе данного хеша.
3. Скачать соответствующий файл `/serversig` и побайтово сравнить его с подписью, сформированной на шаге 2.
4. Рассчитать и проверить (сравнив со страницей `/simple`) хэши MD5 для всех скачиваемых с зеркального сервера файлов.

При скачивании файлов с центрального сервера проверка не требуется и клиенты должны отказаться от ее выполнения с целью снижения нагрузки.

Примерно раз в год ключ меняется на новый. Зеркальные серверы должны повторно получить все страницы `/serversig` после смены ключа. Клиенты, использующие зеркальные серверы, должны получить заслуживающую доверия копию ключа сервера. Одним из способов получения ключа является его загрузка с ресурса <https://pypi.python.org/serverkey>. Для установления факта реализации атак перехвата трафика клиенты должны проверять SSL-сертификат сервера, который должен быть подписан центром CACert.

14.5. Подробности реализации

Реализация большинства описанных в предыдущих разделах улучшений была проведена в рамках проекта `Distutils2`. Файл `setup.py` больше не используется, а проект полностью описывается с помощью файла `setup.cfg`, являющегося статическим `.ini`-подобным файлом. Таким образом мы упростили работу лиц, осуществляющих упаковку проектов, позволив изменять ход установки проекта без вмешательства в код на языке Python. Ниже приведен пример такого файла:

```
[metadata]
name = MPTools
version = 0.1
author = Tarek Ziade
author-email = tarek@mozilla.com
summary = Set of tools to build Mozilla Services apps
description-file = README
home-page = http://bitbucket.org/tarek/pypi2rpm
project-url: Repository, http://hg.mozilla.org/services/server-devtools
classifier = Development Status :: 3 - Alpha
License :: OSI Approved :: Mozilla Public License 1.1 (MPL 1.1)

[files]
packages =
    mopytools
    mopytools.tests

extra_files =
    setup.py
    README
    build.py
    _build.py

resources =
    etc/mopytools.cfg {confdir}/mopytools
```

Пакет `Distutils2` использует файл конфигурации для:

- генерации файлов метаданных `META-1.2`, которые могут использоваться в различных целях, таких, как регистрация проекта в каталоге PyPI.
- выполнения любой команды управления пакетами, такой, как `sdist`.
- установки проекта на основе `Distutils2`.

В рамках проекта `Distutils2` также реализован механизм схем версий `VERSION` с помощью модуля `version`.

Реализация механизма создания списка установленных файлов `INSTALL-DB` будет добавлена в стандартную библиотеку Python версии 3.3 в рамках модуля `pkgutil`. В промежуток времени до добавления в стандартную библиотеку версия этого модуля существует в проекте `Distutils2` и доступна для непосредственного использования. Предоставляемые API позволяют просматривать установки проектов и получать списки установленных файлов.

Эти API являются базисом для некоторых замечательных возможностей, предоставляемых `Distutils2`:

- системы установки/удаления пакетов
- просмотра графов зависимостей для установленных проектов

14.6. Выученные уроки

14.6.1. О стандартах РЕР

Изменение архитектуры таких обширных и сложных проектов, как система управления пакетами Python, должно производиться аккуратно путем изменения стандартов РЕР. А само изменение или добавление нового стандарта РЕР, по моему опыту, занимает около года.

Одной ошибкой, которую совершило сообщество является разработка инструментов, исправляющих некоторые недоработки путем расширения метаданных и изменения хода установки приложений Python, без попытки изменения затронутых стандартов РЕР.

Другими словами, в зависимости от используемого вами инструмента, `Distutils` из стандартной библиотеки или `Setuptools`, приложения устанавливались по-разному. Проблемы были преодолены одной частью сообщества, использующей новые инструменты, но для всего остального мира проблем только добавилось. Лица, ответственные за создание пакетов для операционных систем, столкнулись с несколькими стандартами Python: официальным документированным стандартом и фактически используемым стандартом, установленным разработчиками пакета `Setuptools`.

Но между тем, команда разработчиков пакета `Setuptools` имела возможность проведения экспериментов над огромной аудиторией (всем сообществом), быстро внедряя инновации, поэтому информация от пользователей была бесценна. У нас же была возможность разрабатывать новые стандарты РЕР с большей уверенностью в том, что будет работать, а что не будет и, возможно, по-другому этого сделать бы и не удалось. Таким образом, процесс заключался в наблюдении за тем, какие инновации предлагались сторонними инструментами и решали ли эти инновации проблемы достаточно хорошо, чтобы инициировать изменение стандарта РЕР.

14.6.2. Пакет, добавленный в стандартную библиотеку, находится одной ногой в могиле

Я перефразировал Гвидо ван Россума в названии раздела, но существует один аспект философии "batteries-included" Python, который значительно повлиял на нашу работу.

Пакет `Distutils` является частью стандартной библиотеки и пакет `Distutils2` также скоро станет ее частью. Пакет, являющийся частью стандартной библиотеки, очень сложно развивать. Конечно же, существуют процессы по удалению устаревшего кода, в ходе которых вы можете удалить или изменить часть API после 2 подверсий Python. Но как только происходит публикация API, данный интерфейс остается неизменным в течении многих лет.

Таким образом, любое сделанное вами изменение в пакете стандартной библиотеки, не являющееся исправлением ошибки, является потенциальным нарушением целостности экосистемы. Поэтому тогда, когда вы делаете важные изменения, вы должны создавать новый пакет.

Я убедился в этом на собственном опыте, когда в процессе работы над пакетом `Distutils` мне пришлось убрать все изменения, сделанные мною более чем за год, и создать пакет `Distutils2`. В будущем, если наши стандарты снова изменятся кардинальным образом, велика вероятность того, что мы с самого начала начнем работу над отдельным проектом `Distutils3`, конечно же, если в стандартной библиотеке не появится другой установщик.

14.6.3. Обратная совместимость

Изменение принципа работы системы управления пакетами в Python является очень долгим процессом: экосистема Python содержит множество проектов, базирующихся на устаревших инструментах управления пакетами, поэтому сопротивление изменениям будет весьма значительным. (Достижение соглашения по вопросам, описанным в данной главе книги, заняло несколько лет вместо нескольких месяцев, как я ожидал сначала.) Как и в случае с Python 3, пройдут годы перед тем, как все проекты перейдут на использование нового стандарта.

Именно поэтому все разрабатываемые нами программные продукты должны поддерживать обратную совместимость с используемыми ранее инструментами, установками и стандартами, что делает реализацию пакета `Distutils2` еще более сложной.

Например, если проект, использующий новые стандарты, зависит от другого проекта, который их пока еще не использует, мы не можем прерывать процесс установки, сообщая пользователю о том, что пакет, от которого зависит проект, использует неизвестный формат!

Стоит отметить, что реализация механизма `INSTALL-DB` содержит код для совместимости, позволяющий работать с проектами, установленными с помощью оригинального пакета `Distutils`, `Pip`, `Distribute` или `Setuptools`. `Distutils2` также поддерживает возможность установки проектов, созданных с использованием оригинального пакета `Distutils`, конвертируя метаданные в ходе установки.

14.7. Справочные материалы и вклад сообщества

Некоторые разделы этой главы были взяты напрямую из различных документов PEP, разработанных нами для стандартизации процесса управления пакетами. Вы можете найти оригинальные документы на сайте <http://python.org>:

- PEP 241: Metadata for Python Software Packages 1.0: <http://python.org/peps/pep-0214.html>
- PEP 314: Metadata for Python Software Packages 1.1: <http://python.org/peps/pep-0314.html>
- PEP 345: Metadata for Python Software Packages 1.2: <http://python.org/peps/pep-0345.html>
- PEP 376: Database of Installed Python Distributions: <http://python.org/peps/pep-0376.html>
- PEP 381: Mirroring infrastructure for PyPI: <http://python.org/peps/pep-0381.html>
- PEP 386: Changing the version comparison module in Distutils: <http://python.org/peps/pep-0386.html>

Я хотел бы поблагодарить всех людей, которые работали над системой управления пакетами; вы можете найти их имена в любом из документов PEP, о которых я упоминал. Я также хотел бы особо поблагодарить всех участников Товарищества Разработчиков Системы Пакетов. Также благодарю Alexis Maitaineau, Toshio Kuratomi, Holger Krekel и Stefane Fermigier за их отзывы к данной главе.

Проекты, которые мы обсуждали в данной главе:

- `Distutils`: <http://docs.python.org/distutils>
- `Distutils2`: <http://packages.python.org/Distutils2>
- `Distribute`: <http://packages.python.org/distribute>
- `Setuptools`: <http://pypi.python.org/pypi/setuptools>
- `Pip`: <http://pypi.python.org/pypi/pip>

- Virtualenv: <http://pypi.python.org/pypi/virtualenv>

Сноски

1. Предложения по изменению в Python (The Python Enhancement Proposals - PEP), на которые мы ссылаемся, описаны в конце этой главы.
2. Ранее известный как CheeseShop.
3. Т.е. RFC 3280 SubjectPublicKeyInfo с алгоритмом 1.3.14.3.2.12.
4. Т.е. как RFC 3279 Dsa-Sig-Value, на основе алгоритма 1.2.840.10040.4.3.

15. Riak и Erlang/OTP

Глава 15 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

Riak является распределенной отказоустойчивой СУБД с открытым исходным кодом, на которой проиллюстрировано, как с помощью среды Erlang/OTP создавать крупномасштабные системы. Во многом благодаря тому, что в языке Erlang поддерживается работа с масштабируемыми распределенными системами, в Riak предлагаются функции, которые весьма редки в базах данных, например, высокая готовность и линейная масштабируемость, причем как по емкости базы, так и по ее пропускной способности.

Erlang/OTP является идеальной платформой для разработки таких систем, как Riak, поскольку в ней сразу «из коробки» предлагаются средства взаимодействия между узлами, очереди сообщений, детекторы отказов и клиент-серверные абстракции. Более того, наиболее часто используемые в языке Erlang шаблоны реализованы в виде библиотечных модулей, которые обычно имеют в виду, когда говорят о поведениях среды OTP (OTP behaviors). В них содержится основной фреймворк кода, предназначенный для параллельной работы и обработки ошибок, что упрощает параллельное программирование и защищает разработчика от многих распространенных ошибок. Поведения контролируются супервизорами, самим поведением, и все они сгруппированы в виде дерева мониторинга. Дерево мониторинга упаковывается в приложение, представляющее собой строительный блок программы на языке Erlang.

Полная система Erlang, например, Riak представляет собой набор слабо связанных приложений, которые взаимодействуют друг с другом. Некоторые из этих приложений пишутся разработчиком, некоторые из них являются частью стандартного дистрибутива Erlang/OTP, а некоторые могут быть другими компонентами, имеющими открытый исходный код. Они последовательно загружаются и запускаются с помощью загрузочного скрипта, созданного из списка приложений и версий.

Системы различаются своими приложениями, которые являются частью запускаемого релиза. В стандартном дистрибутиве Erlang загрузочные файлы будут запускать ядро *Kernel* и *StdLib* (стандартная библиотека). В некоторых инсталляциях также запускается приложение *SASL* (Systems Architecture Support Library — Библиотека поддержки архитектуры систем). В SASL содержится релиз и инструментарий обновления программ, а также средства регистрации событий. Riak ничем не отличается, кроме запуска конкретных для Riak приложений, а также их зависимостей времени выполнения, к числу которых относятся *Kernel*, *StdLib* и *SASL*. Эти стандартные элементы дистрибутива Erlang/OTP входят, на самом деле, в состав полной и готовой для запуска сборки Riak, и, когда из командной строки вызывается команда `riak start`, они все запускаются в унисон. Riak состоит из многих сложных приложений, поэтому эту главу не нужно рассматривать в качестве полного руководства. Ее следует рассматривать как введение в OTP, в котором в качестве примеров используется исходный код Riak. Для удобства демонстрации рисунки и примеры кода были упрощены и сокращены.

15.1. Краткое введение в язык Erlang

Язык Erlang является функциональным языком параллельного программирования, который компилируется в байт-код и работает в виртуальной машине. Программы состоят из функций, которые вызывают друг друга и часто возвращают результаты в виде межпроцессорных сообщений, представляющих собой побочный эффект, а также выполняют операции ввода вывода и обращаются к базе данных. Значения переменным языка Erlang присваиваются только один раз, то есть, если им было присвоено значение, оно не может быть изменено. В языке широко используется сравнение с шаблонами так, как это показано ниже на примере расчета факториала:

```
-module(factorial).
-export([fac/1]).
fac(0) -> 1;
fac(N) when N>0 ->
    Prev = fac(N-1),
    N*Prev.
```

Здесь первое предложение (clause) выдает факториал нуля, второе — выдает факториал положительных чисел. Тело каждого оператора представляет собой последовательность выражений, и последнее выражение в теле является результатом этого предложения. Вызов функции с отрицательным числом приведет к ошибке времени выполнения, поскольку предложение сравнения для такого случая отсутствует. Отсутствие обработки этого случая является примером небезопасного программирования, практика которого поощряется в языке Erlang.

Внутри модуля обращение к функции осуществляется обычным образом; вне модуля — сначала добавляется имя модуля, например, `factorial:fac(3)`. Можно определять функции с тем же самым именем, но различным числом аргументов, это называется их *арностью*. В директиве `export` в модуле `factorial` арность функция `fac` равна единице, что обозначено как `fac/1`.

В языке Erlang поддерживается работа с кортежами (также называемыми продукционными типами), а также со списками. Кортежи заключаются в фигурные скобки, например, `{ok, 37}`. В кортежах мы получаем доступ к элементам по их позиции. Записи являются другим типом данных, они позволяют нам хранить фиксированное количество элементов, доступ к которым и работа с ним осуществляется по имени. Мы определяем запись при помощи директивы `-record(state, {id, msg_list=[]})`. Чтобы создать экземпляр, мы используем выражение `Var = #state{id=1}`, и мы проверяем его содержимое при помощи `Var#state.id`. Если число элементов переменное, мы используем списки, которые записываются в квадратных скобках, например, `{[]|23, 34|[]}`. Нотация `{[]|Xs{}}` соответствует непустому списком с головой `X` и хвостом `Xs`. Идентификаторы, начинающиеся с маленькой буквы, обозначают атомы, которые просто представляют самих себя; `ok` в кортеже `{ok, 37}` является примером атома. Атомы, используемые таким образом, часто применяются для указания различных видов результата функции: точно также как результаты `ok`, могут быть результаты вида `{error, "Error String"}`.

Процессы в системах Erlang работают параллельно в отдельно выделяемых областях памяти и взаимодействуют между собой посредством передачи сообщений. Процессы могут использоваться в приложениях для всего, в том числе в виде шлюзов к базам данных, обработчиков стеков протоколов, а также для управления средствами регистрации сообщений трассировки других процессов. Хотя эти процессы обрабатывают различные виды запросов, они аналогичны в том, как эти запросы обрабатываются.

Поскольку процессы существуют только в виртуальной машине, одна виртуальная машина может одновременно выполнять миллионы процессов — это та особенность Riak, которой широко пользуются. Например, каждый запрос к базе данных — чтение, запись и удаление — моделируется как отдельный процесс, это тот подход, который в большинстве случаев нельзя реализовать с помощью работы с потоками на уровне ОС.

Процессы идентифицируются при помощи идентификаторов процессов, которые называются PID, но их также можно зарегистрировать с использованием алиаса; этим следует пользоваться только

для долгоживущих "статических" процессов. Регистрация процесса с использованием алиаса позволяет другим процессам посыпать ему сообщений, не зная его идентификатора PID. Процессы создаются с помощью встроенной функции `spawn(Module, Function, Arguments)` (`built-in function -BIF`). Функции BIF интегрированы в виртуальную машину и используются для того, чтобы делать то, что в чистом языке Erlang сделать нельзя или что делается слишком медленно. Встроенная функция `spawn/3` получает в качестве параметров модуль `Module`, функцию `Function` и список аргументов `Arguments`. Вызов возвращает идентификатор PID только что порожденного процесса и, в качестве побочного эффекта, создает новый процесс, в котором начинается выполнение функции в модуле с аргументами, упомянутыми ранее.

Сообщение `Msg` отправляется в процесс с идентификатором `Pid` при помощи конструкции `Pid ! Msg`. Идентификатор PID процесса можно узнать, вызвав встроенную функцию `self`, и затем его можно отослать в другие процессы для того, чтобы они использовали его для обмена сообщениями с исходным процессом. Предположим, что процесс ожидает получения сообщений вида `{ok, N}` и `{error, Reason}`. Чтобы их обработать, используется следующая инструкция приема сообщений:

```
receive
    {ok, N} ->
        N+1;
    {error, _} ->
        0
end
```

Результатом работы этой инструкции является число, определяемое в предложении, в котором происходит сравнения с образцом. Если в сравнении с образцом не требуется определять значение переменной, то можно использовать универсальный символ подчеркивания так, как это показано выше.

Передача сообщений между процессами является асинхронной, и сообщения, принимаемые процессом, помещаются в почтовый ящик процесса в том порядке, в котором они поступают. Теперь предположим, что выражение `receive`, приведенное выше, выполняется: если первый элемент в почтовом ящике будет `{ok, N}` или `{error, Reason}`, то будет возвращен соответствующий результат. Если первое сообщение в почтовом ящике имеет другой вид, то оно сохраняется в почтовом ящике, и таким же самым образом обрабатывается второе сообщение. Если нет совпадающих сообщений, то выражение `receive` будет ждать до тех пор, пока не поступит совпадающее сообщение.

Процессы завершаются по двум причинам. Если для выполнения больше нет кода, то говорят, что процесс завершается по причине *normal* (нормальное завершение). Если в процессе во время выполнения возникают ошибки, то говорят, что он завершается по причине *non-normal* (ненормальное завершение). Завершение процесса не влияет на другие процессы, если они с ним не скомпонованы. Процессы можно скомпоновать друг с другом с помощью встроенной функции `link(Pid)` или при вызове `spawn_link(Module, Function, Arguments)`. Если процесс завершается, то он посылает сигнал выхода EXIT в процессы, с которыми он скомпонован. Если происходит ненормальное завершение, то процесс завершается самостоятельно перенаправляя дальше сигнал EXIT. Если вызвать встроенную функцию `process_flag(trap_exit, true)`, то вместе завершения процессы могут получить в своих почтовых ящиках сигналы EXIT, представляющие собой сообщения языка Erlang.

В Riak сигналы EXIT используются для мониторинга нормального состояния вспомогательных процессов обработки некритических операций, инициированных по запросам конечных автоматов. Если эти вспомогательные процессы завершаются ненормально, сигнал EXIT позволяет родительскому процессу либо игнорировать ошибку, либо перезапустить процесс.

15.2. Остов процесса

Ранее мы определили, что процессы создаются по общему образцу независимо от того, для какой цели они были созданы. Для начала, должен быть создан процесс, а затем, при необходимости, должен быть зарегистрирован алиас. Первым действием вновь созданного процесса является инициализации данных цикла процесса. Данные цикла часто создаются в результате передачи аргументов при инициализации процесса во встроенную функцию `spawn`. Данные цикла процесса сохраняются в переменной, которую мы называем состоянием процесса. Состояние, часто сохраняемое в виде записи, передается в функцию приема-оценки,ирующую в цикле, который получает сообщение, обрабатывает его, обновляет состояние, и передает его обратно в качестве аргумента в вызов оставшейся части рекурсивного обработки. Если одно из сообщений, которые он обрабатывает, является сообщением «`stop`», то процесс, принявший его, очистит все за собой, а затем завершится.

Это повторяющаяся тема, которая будет происходить с процессами независимо от задачи, выполняемой процессом. Помня об этом, давайте посмотрим на различия между процессами, соответствующими этому образцу:

- Аргументы, передаваемые в вызовы встроенной функции `spawn`, будут в различных процессах различными.
- Вы должны решить, должны ли вы регистрировать процесс с алиасом, и если вы будете регистрировать, то должны решить, какой алиас следует использовать.
- В функции, инициализирующей состояние процесса, выполняемые действия различаются в зависимости от задач, которые будет выполнять процесс.
- Состояние системы в каждом случае представлено данными цикла, но содержимое данных цикла будет варьироваться в зависимости от процессов.
- В теле, в котором осуществляется прием и оценка данных цикла, процессы будут получать различные сообщения и обрабатывать их по-разному.
- И, наконец, очистка данных, выполняемая при завершении процесса, будет варьироваться и будет зависеть от конкретного процесса.

Поэтому, даже если и есть скелет общей последовательности действий, эти действия дополняются другими действиями, которые непосредственно связаны с конкретными задачами, выполняемыми в этом процессе. Используя этот скелет в качестве шаблона, программисты могут создавать процессы языка Erlang, которые действуют как серверы, конечные автоматы, обработчики событий и супервизоры. Но для того, чтобы каждый раз не реализовывать повторно эти образцы, они были помещены в библиотечные модули, которые называются поведением (behavior). Они поставляются как часть промежуточной среды OTP.

15.3. Поведения OTP

Основной костяк команды разработчиков Riak разбросан почти по десятку различных географических точек. Без очень жесткой координации и шаблонов, по которым должна строиться работа, результат мог бы представлять собой разнообразные клиент/серверные реализации, в которых не обрабатывались специальные граничные случаи и которые бы не справились с ошибками, касающимися параллельной обработки. Вероятно, нет единого способа работы с клиент/серверными сбоями, и нет гарантии, что ответ на запрос, действительно является ответом, а не просто некоторым сообщением, не противоречащим внутреннему протоколу передачи сообщений.

OTP представляет собой набор библиотек Erlang и принципов проектирования, предлагаемых в виде набора готовых к применению инструментальных средств, предназначенных для разработки надежных систем. Многие из этих образцов и библиотек предоставлены в виде «поведений» («behaviors»).

Поведения ОТР решают эти вопросы путем предоставления библиотечных модулей, реализующих наиболее распространенные шаблонов параллельного проектирования. В самой библиотеке в фоновом режиме реализуется, причем так, что программист об этом может не знать, постоянная обработка ошибок и специальных случаев. В результате, поведения ОТР предоставляют набор стандартных блоков, используемых при проектировании и создании систем промышленного класса.

15.3.1. Введение

Поведения OTP представлены в виде библиотечных модулей в приложении `stdlib`, которое поставляется как часть дистрибутива Erlang/OTP. Конкретный код, написанный программистом, помещается в отдельный модуль и вызывается через набор стандартных предопределенных функций обратного вызова, которые стандартизированы для каждого поведения. Такой модуль обратного вызова будет содержать весь конкретный код, необходимый для достижения требуемой функциональности.

Поведения ОТР включают в себя рабочие процессы (worker process), которые выполняют фактическую обработку, и супервизоры (supervisor), задачей которых является наблюдение за рабочими процессами и другими супервизорами. Поведения рабочих процессов, часто обозначаемые в диаграммах в виде кружков, включают в себя серверы, обработчики событий и конечные автоматы. Супервизоры, обозначаемые на иллюстрациях в виде прямоугольников, следят за дочерними элементами, причем как за рабочими процессами, так и за другими супервизорами, создавая на то, что называется деревом мониторинга.

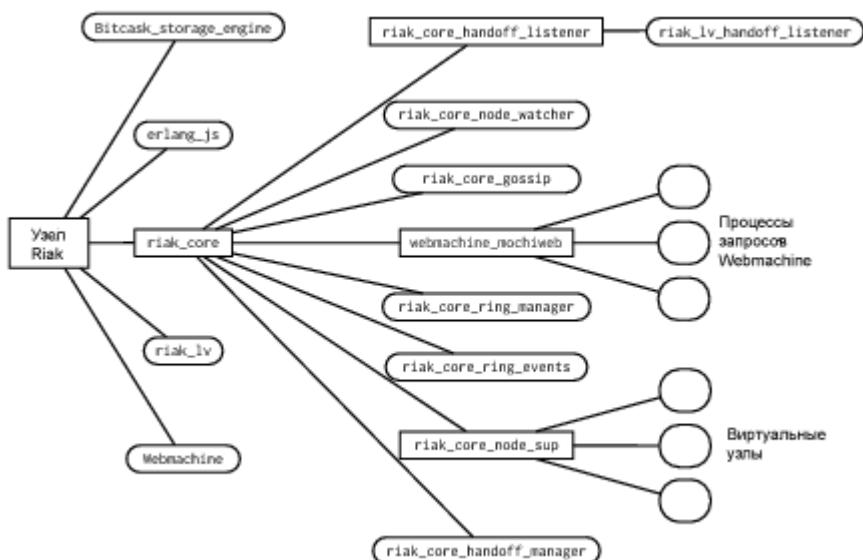


Рис.15.1: Дерево мониторинга OTP Riak

Деревья мониторинга упакованы в поведение, которое называется приложением. Приложения OTP не только являются строительными блоками систем Erlang, но также и способом упаковки повторно используемых компонентов. Системы промышленного уровня, такие как Riak, состоят из множества слабо связанных, возможно, распределенных приложений. Некоторые из этих приложений являются частью стандартного дистрибутива Erlang, а некоторые являются теми частями, в которых реализована конкретная функциональность Riak.

К числу примеров приложений OTP относятся CORBA ORB или агент Simple Network Management Protocol (SNMP). Приложение OTP является повторно используемым компонентом, в котором упакованы библиотечные модули вместе с супервизорами и рабочими процессами. Теперь, когда мы говорим о приложении, мы будем подразумевать приложение OTP.

Модули поведения содержат весь общий код для каждого конкретного типа поведения. Хотя можно реализовать свой собственный модуль поведения, делается это редко, поскольку те модули, ко-

торые приходят с дистрибутивом Erlang/OTP, обычно содержат большую часть проектных шаблонов, которые вам может потребоваться использовать в своем коде. К числу общих функциональных возможностей, которые предоставлены в модуле поведения, относятся следующие операции:

- порождение и, возможно, регистрация процесса;
- отправка и получение клиентских сообщений, как с помощью синхронных, так и асинхронных вызовов, в том числе определяющих внутренний протокол обмена сообщениями;
- запоминание данных цикла и управления циклом процесса, и
- остановка процесса.

Данные цикла являются переменной, в которой поведение будет хранить данные, необходимые ему между вызовами. После вызова будет возвращаться обновленный вариант данных цикла. Эти обновленные данные цикла, которые часто называются новыми данными цикла, передаются в качестве аргумента в следующий вызов. Данные цикла также часто называются состоянием поведения.

К числу функциональных возможностей, которые должны быть в модуле обратного вызова для общего сервера приложений с тем, чтобы в нем реализовать требуемое поведение, относятся следующие:

- Инициализация данных цикла процесса, и, если процесс зарегистрирован, имени процесса.
- Обработка конкретных клиентских запросов и, если есть синхронизация, отсылка ответов обратно клиенту.
- Обработка и обновление данных цикла процесса, осуществляемые между запросами процесса.
- Очистка данных цикла процесса по окончанию.

15.3.2 Основные серверы

Основные серверы, в которых реализовано поведение клиент/сервер, определены в поведении `gen_server`, которое поставляется как часть стандартного библиотечного приложения. При рассмотрении основных серверов, мы будем пользоваться модулем `riak_core_node_watcher.erl` из приложения `riak_core`. Это сервер, который отслеживает и сообщает о том, какие суб-сервисы и узлы имеются в кластере Riak. Заголовки и директивы модуля выглядят следующим образом:

```
-module(riak_core_node_watcher).
-behavior(gen_server).
%% API
-export([start_link/0, service_up/2, service_down/1, node_up/0, node_down/0, services/0,
        services/1, nodes/1, avsn/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
        code_change/3]).

-record(state, {status=up, services=[], peers=[], avsn=0, bcast_tref,
               bcast_mod={gen_server, abcast}}).
```

Мы можем легко отличить общий сервер по директиве `-behavior(gen_server)`. Эта директива используется компилятором для того, чтобы правильно экспортовать все необходимые функции обратного вызова. В данных цикла сервера используется информация о статусе записи.

15.3.3. Запуск вашего сервера

Когда используется поведение `gen_server`, вы вместо встроенных функций `spawn` и `spawn_link` будете пользоваться функциями `gen_server:start` и `gen_server:start_link`. Основным различием между `spawn` и `start` является синхронизированная природа выполнения вызова. Использо-

вание `start` вместо `spawn` делает запуск рабочего процесса более детерминированным и предотвращает возникновение непредвиденных состояний гонки (*race conditions*), поскольку вызов не вернет идентификатор PID рабочего процесса до тех пор, пока процесс не будет инициализирован. Вы вызываете функции одним из следующих способов:

```
gen_server:start_link(ServerName, CallbackModule, Arguments, Options)
gen_server:start_link(CallbackModule, Arguments, Options)
```

`ServerName` является кортежем вида `{local, Name}` или `{global, Name}`, в котором указывается локальное или глобальное имя `Name` алиаса процесса для случая, если имя должно быть зарегистрировано. Глобальные имена позволяют серверам прозрачно обращаться в кластеры распределенных узлов Erlang. Если вы не хотите регистрировать процесс и вместо этого ссылаетесь на него по его идентификатору PID, вы не указываете имя и вместо этого пользуетесь функцией `start_link/3` или `start/3`. `CallbackModule` является именем модуля, в котором размещены конкретные функции обратного вызова, `Arguments` является допустимым термином языка Erlang, который передается в функцию обратного вызова `init/1`, а `Options` является списком, с помощью которого вы сможете устанавливать флаги `fullsweep_after` и `heapsize`, а также другие флаги, используемые при трассировке и отладке.

В нашем примере, мы вызываем `start_link/4` и с помощью макроподстановки `?MODULE`, регистрируя процесс с тем же именем, что и у модуля обратного вызова. В этом макросе указывается имя модуля, которое будет определено препроцессором во время компиляции кода. Рекомендуется всегда задавать имя вашего поведения точно таким же, как и имя модуля обратного вызова, в котором оно реализовано. Мы не передаем никаких аргументов, и, в результате, просто отправляем пустой список. Список параметров остается пустым:

```
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

Очевидным различием между функциями `start_link` и `start` является то, что `start_link` скомпонована со своим родителем, которым чаще всего оказывается супервизором, тогда как для `start` этого не происходит. Об этом необходимо особо упомянуть, поскольку поведение OTP само должно ссылаться на супервизор. Функции `start` часто используются при тестировании поведения внутри командной оболочки, т. к. ошибки ввода, из-за которых возникает сбой работы командной оболочки, не оказывают влияние на поведение. Все варианты функций `start` и `start_link` возвращают `{ok, Pid}`.

Функции `start` и `start_link` порождают новый процесс, в котором будет вызвана функция обратного вызова `init(Arguments)`, находящаяся в модуле `CallbackModule`, с аргументами `Arguments`. Функция `init` должна инициализировать данные цикла сервера `LoopData` и должен возвращаться кортеж вида `{ok, LoopData}`. В данных `LoopData` содержится первый экземпляр цикла данных, которые будут передаваться между функциями обратного вызова. Если вы хотите сохранить некоторые из аргументов, передаваемые вами в функцию `init`, вы их также должны сохранять в переменной `LoopData`. Данные `LoopData` в сервере-наблюдателе узла Riak являются результатом работы функции `schedule_broadcast/1`, вызываемой с записью типа `state`, в которой значения, используемые по умолчанию, устанавливаются в полях этой записи следующим образом:

```
init([]) ->
    %% Watch for node up/down events
    net_kernel:monitor_nodes(true),
    %% Setup ETS table to track node status
    ets:new(?MODULE, [protected, named_table]),
    {ok, schedule_broadcast(#state{})}.
```

Хотя процесс супервизора может вызвать функция `start_link/4`, другой процесс вызывает функцию обратного вызова `init/1`: это тот процесс, который был только что создан. Поскольку назначение этого сервера обнаруживать, записывать и передавать всем сообщения о наличии внутри Riak подсервисов, во время инициализации делается запрос среди времени исполнения языка Erlang обнаруживать такие события и настраивается таблица, в которой запоминается эта информация. Это нужно делать во время инициализации, поскольку в случае, если этой структуры еще нет, любые обращения к серверу окончатся неудачей. В вашей функции `init` делайте только то, что необходимо, и минимизируйте количество операций, поскольку вызов функции `init` является синхронным вызовом, который не позволит запустить какой-нибудь другой процесс сериализации до тех пор, пока управление не будет возвращено из этой функции.

15.3.4. Передача сообщений

Если вы хотите отправить синхронное сообщение на ваш сервер, вы используете функцию `gen_server:call/2`. Асинхронные вызовы выполняются с помощью функции `gen_server:cast/2`. Давайте начнем рассмотрение с этих двух функций API, относящихся к сервисам приложения Riak; остальной код мы рассмотрим позже. Эти функции вызываются клиентским процессом и результатом является синхронное сообщение, посыпанное серверному процессу, зарегистрированному с тем же самым именем, что и модуль обратного вызова. Обратите внимание, что проверка данных, передаваемых на сервер, должна происходить на клиентской стороне. Если клиент посылает неверную информацию, сервер должен завершить свою работу.

```
service_up(Id, Pid) ->
    gen_server:call(?MODULE, {service_up, Id, Pid}).

service_down(Id) ->
    gen_server:call(?MODULE, {service_down, Id}).
```

После получения сообщения, процесс `gen_server` вызывает функцию обратного вызова `handle_call/3`, получающую сообщения в том же самом порядке, в котором они были отправлены:

```
handle_call({service_up, Id, Pid}, _From, State) ->
    %% Update the set of active services locally
    Services = ordsets:add_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% Remove any existing mrefs for this service
    delete_service_mref(Id),

    %% Setup a monitor for the Pid representing this service
    Mref = erlang:monitor(process, Pid),
    erlang:put(Mref, Id),
    erlang:put(Id, Mref),

    %% Update our local ETS table and broadcast
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};

handle_call({service_down, Id}, _From, State) ->
    %% Update the set of active services locally
    Services = ordsets:del_element(Id, State#state.services),
    S2 = State#state { services = Services },

    %% Remove any existing mrefs for this service
    delete_service_mref(Id),

    %% Update local ETS table and broadcast
    S3 = local_update(S2),
    {reply, ok, update_avsn(S3)};
```

Обратите внимание на значение, возвращаемое функцией обратного вызова. В кортеже содержится управляющий атом `reply`, сообщающий общему коду `gen_server` о том, что второй элемент кортежа (который в обоих наших случаях является атомом `ok`) является ответом, отправляемым обратно к клиенту. Третий элемент кортежа является новым состоянием `new State`, которое, в новой итерации сервера, передается в качестве третьего аргумента в функцию `handle_call/3`; в обоих случаях оно здесь обновляется с целью отобразить новое состояние имеющихся сервисов. Аргумент `_From` является кортежем, содержащим уникальную ссылку на сообщение и идентификатор клиентского процесса. Кортеж, как целое, используется в библиотечных функциях, которые мы в этой главе рассматривать не будем. В большинстве случаев, вам он не понадобится.

Библиотечный модуль `gen_server` имеет ряд встроенных механизмов и средств защиты, действующими за кулисами. Если ваш клиент посыпает синхронное сообщение на ваш сервер, и вы не получаете ответ в течение пяти секунд, то процесс выполнения функции `call/2` завершается. Вы можете изменить это, использовав для этого `gen_server:call(Name, Message, Timeout)`, где `Timeout` является значением, указываемым в миллисекундах, или атомом бесконечности `infinity`.

Механизм тайм-аута сначала был добавлен для того, чтобы предотвратить взаимной блокировки и гарантировать, что работа серверов, которые случайно вызовут друг друга, будет завершена после тайм-аута, заданного по умолчанию. Сообщение об аварийной ситуации будет зарегистрировано, и, можно надеяться, приведет к тому, что ошибка будет найдена и исправлена. При тайм-ауте в пять секунд большинство приложений будут работать так, как это необходимо, но при очень больших нагрузках, вам, возможно, придется более точно задавать это значение или, возможно, даже использовать значение бесконечности `infinity`; этот выбор зависит от приложения. Во всех фрагментах с критическим кодом в Erlang/OTP используется `infinity`. В различных местах в Riak используются различные значения тайм-аута: `infinity` обычно используется в случае связанных между собой частей кода, а значения `Timeout` устанавливаются с учетом значения параметра, передаваемого пользователем, там, где в клиентском коде в Riak передается информация о том, что у операции должна быть возможность использовать тайм-аут.

Другие защитные механизмы, применяемые в случае использования функции `gen_server:call/2`, требуются в случае отправки сообщения на несуществующий сервер и в случае, когда сбой сервера происходит раньше, чем он отправит свой ответ. В обоих случаях, вызывающий процесс будет завершен. В чистом языке Erlang отправка сообщения, для которого в принимающем предложении никогда не происходит совпадения с образцом, рассматривается как ошибка, которая может привести к утечке памяти. Чтобы смягчить эту ситуацию, в Riak применяются две различные стратегии, использующих предложения сравнения, в которых сравнение осуществляется «со всем». В местах, где сообщение может быть инициировано пользователем, сообщение, которое не прошло сравнение, может быть просто отброшено. В местах, где такое сообщение может поступать только изнутри Riak, оно представляет собой ошибку, и поэтому будет выдан рапорт о внутреннем сбое, вызванном ошибкой, и рабочий процесс, который получил это сообщение, будет перезапущен.

Отправка асинхронных сообщений работает аналогичным образом. Сообщения отправляются асинхронно в общий сервер и обрабатываются с помощью функция обратного вызова `handle_cast/2`. Функция должна возвращать кортеж вида `{reply, NewState}`. Асинхронные вызовы используются, когда нас не интересует запрос сервера и мы не беспокоимся о том, что отправляем больше сообщений, чем сервер может принять. В случаях, когда нас не интересует ответ, но мы хотим перед тем, как отослать следующее сообщение, подождать, пока не будет обработано первое сообщение, нам нужно использовать `gen_server:call/2`, которое в ответе возвратит атом `ok`. Представьте себе процесс создания записей базы данных, который происходит быстрее, чем может принять Riak. Если мы используем асинхронные вызовы, мы рискуем заполнить почтовый ящик процесса и создать в узле состояние нехватки памяти. Riak для регулировки нагрузки пользуется возможностью сериализации сообщений синхронных вызовов `gen_server`, при котором обработка следующего запроса происходит только после того, как был обработан предыдущий запрос. Такой подход устраняет необходимость в более сложных схемах управления кодом: в доба-

вок тому, что процессы `gen_server` позволяют осуществлять параллельную обработку, их также можно использовать для выполнения сериализации.

15.3.5. Остановка сервера

Как остановить сервер? Вы можете в функциях обратного вызова вместо возврата кортежа `{reply, Reply, NewState}` или `{noreply, NewState}`, возвратить кортеж `{stop, Reason, Reply, NewState}` или `{stop, Reason, NewState}`, соответственно. Что-то должно быть причиной возврата такого значения — часто это сообщение об остановке, отправляемое на сервер. После получения кортежа, сообщающего об остановке и указывающего причину `Reason` и состояние `State`, общий код выполнит функцию обратного вызова `terminate(Reason, State)`.

Функция `terminate` является обычно тем местом, куда вставляется код, необходимый для очистки состояния сервера `State` и любых других постоянно хранящихся данных, используемых системой. В нашем примере мы посылаем последнее сообщение всем нашим процессам, наблюдающим за этим узлом, с тем, чтобы они знали, что этот узел будет остановлен. В этом примере в переменной `State` содержится запись с полями состояний `status` и наблюдателей `peers`:

```
terminate(_Reason, State) ->
    %% Let our peers know that we are shutting down
    broadcast(State#state.peers, State#state{status = down}).
```

Использование функций обратного вызова поведения в качестве библиотечных функций и их вызов из других частей программы является очень плохим практическим примером. Например, для того, чтобы получить исходные данные цикла, вы никогда не должны вызывать `riak_core_node_watcher:init(Args)` из другого модуля. Такое получение данных должно осуществляться с помощью синхронного обращения к серверу. Обращения к функциям обратного вызова поведения должно происходить из библиотечных модулей только в случае событий, возникающих в самой системе, а не напрямую пользователем.

15.4. Другие поведения рабочих процессов

С помощью тех же самых идей можно реализовывать и были реализованы много других видов поведений рабочих процессов.

15.4.1. Автоматы конечных состояний

Конечные автоматы (или машины с конечным числом состояний - FSM), реализованные в модуле поведения `gen_fsm`, являются важнейшим компонентом реализации стеков протоколов, используемых в сетях связи (той проблемной области, для которой первоначально и был создан язык Erlang). Состояния определяются как функции обратного вызова, в названиях которых отражено то, что возвращается в кортеже, имеющем переменную следующего состояния `State`, и то, каким образом обновляются данные цикла. Вы можете отправлять события для этих состояний как синхронно, так и асинхронно. Модуль функции обратного вызова конечного автомата также должен иметь возможность экспорттировать стандартные функции обратного вызова, например, `init`, `terminate` и `handle_info`.

Разумеется, конечные автоматы не предназначены исключительно для целей телекоммуникаций. В Riak, они используются в обработчиках запросов. Когда клиент выдает запрос, например, `get`, `put` или `delete`, процесс, услышавший этот запрос, создаст процесс, реализующий соответствующее поведение `gen_fsm`. Например, `riak_kv_get_fsm` отвечающий за обработку запросов `get`, извлекает данные и отправляет их клиентскому процессу. Процесс FSM будет проходить через различные состояния, поскольку в нем определено, из каких узлов запрашивать данные, как в эти узлы отправлять сообщения, и как в ответ принимать данные, ошибки или тайм-ауты.

15.4.2. Обработчики событий

Обработчики и менеджеры событий являются еще одним поведением, реализованным в библиотечном модуле `gen_event`. Идея состоит в том, чтобы создать централизованное место, где принимаются события определенного вида. События могут передаваться синхронно и асинхронно с заранее определенным набором действий, выполняемых при их получении. Возможными реакциями на события может быть запись их в файл, посылка сообщения об аварии в виде SMS или сбор статистических данных. Каждое из этих действий определяется в отдельном модуле обратного вызова с его собственными данными цикла, которые сохраняются между вызовами. Для каждого конкретного менеджера событий можно добавлять, удалять или обновлять обработчики событий. Таким образом, на практике, в каждом менеджере событий может быть много модулей обратного вызова, а в различных менеджерах событий также может быть много различных экземпляров таких модулей обратного вызова. К числу обработчиков событий относятся процессы, получающие сигналы тревоги, отслеживающие данные в реальном времени, следящие за событиями, связанными с оборудованием, или просто регистрирующие данные в журнале.

Одно из применений в Riak поведения `gen_event` является управление подписками в «кольце событий», т. е. изменений принадлежности узлам разделов или назначение разделов узлам в кластере Riak. Процессы в узле Riak могут регистрировать функцию в экземпляре событий `riak_core_ring_events`, реализующем поведение `gen_event`. Всякий раз, когда центральный процесс, управляющий кольцом событий в этом узле, изменяет запись о принадлежности, он создает событие, в результате которого в каждом из этих модулей обратного вызова будет вызвана зарегистрированная функция. Таким образом, можно достаточно просто организовать реагирование различных частей Riak на изменения в одной из самых центральных структур данных Riak, причем не добавляя сложности в сам механизм централизованного управления этой структурой.

Наиболее распространенные модели распараллеливания и взаимодействия процессов обрабатываются с помощью трех основных типов поведения, которые мы только что рассмотрели: `gen_server`, `gen_fsm` и `gen_event`. Тем не менее, в больших системах с течением времени возникает необходимость в некоторых моделях, предназначенных для конкретного приложения, , т. е. нужно создавать новые виды поведений. В Riak есть одно такое поведение, `riak_core_vnode`, в котором formalизована реализация виртуальных узлов. Виртуальные узлы являются первичной абстракцией хранения данных в Riak, которая по запросам, управляемым с помощью конечных автоматов, предоставляет единый интерфейс хранилища данных вида «ключ - значение». Интерфейс модулей обратного вызова задается с помощью функции `behavior_info/1` следующим образом:

```
behavior_info(callbacks) ->
  [{init,1},
   {handle_command,3},
   {handoff_starting,2},
   {handoff_cancelled,1},
   {handoff_finished,2},
   {handle_handoff_command,3},
   {handle_handoff_data,2},
   {encode_handoff_item,2},
   {is_empty,1},
   {terminate,2},
   {delete,1}];
```

В приведенном выше примере показана функция `behavior_info/1` узла `riak_core_vnode`. В списке кортежей `{CallbackFunction, Arity}` определен контракт, которому должны следовать модули обратного вызова. Эти функции должны быть экспортированы в конкретные реализации виртуальных узлов, иначе компилятор выдаст предупреждение. Реализация своего собственного поведения OTP относительно проста. Кроме определений ваших собственных функций обратного вызова, использующих модули `proc_lib` и `sys`, вам потребуется запускать их вместе с конкрет-

ными функциями, обрабатывать системные сообщения и следить за тем, не завершился ли родительский процесс.

15.5. Супервизоры

Задачей поведения супервизора является отслеживание его потомков и, основываясь на некоторых предварительно заданных правилах, выполнение конкретный действий в случае, когда выполнение потомков завершается. В качестве потомков могут быть как супервизоры, так и рабочие процессы. Это позволило при разработке кода Riak сосредоточить внимание на его корректности, благодаря чему с помощью супервизоров можно во всей системе постоянно следить за ошибками в программном обеспечении, повреждениями данных или системными ошибками. В мире Erlang такой небезопасный подход к программированию часто называют стратегией «пускай выходит из строя». Среди потомков, за которыми наблюдает супервизор, могут быть как супервизоры, так и рабочие процессы. Рабочие процессы являются поведениями OTP, в том числе `gen_fsm`, `gen_server` и `gen_event`. Команда разработчиков Riak, у которой не было возможности обрабатывать пограничные ошибочные ситуации, ограничилась работой с кодом небольшого объема. Размер этого кода из-за того, что в нем используются поведения, гораздо меньшего размера, т. к. это код конкретного приложения. В Riak точно также как и в большинстве приложений Erlang, есть супервизор верхнего уровня, а также есть супервизоры следующих уровней для групп процессов соответствующего назначения. Примерами являются виртуальные узлы Riak, процессы, слушающие сокеты TCP, а также менеджеры запросов-ответов.

15.5.1. Функции обратного вызова супервизора

Чтобы продемонстрировать, как реализуется поведение супервизора, мы воспользуемся модулем `riak_core_sup.erl`. Супервизор ядра Riak является супервизором верхнего уровня приложения ядра Riak. Он запускает набор статических рабочих процессов и супервизоров, а также ряд динамических рабочих процессов, осуществляющих обработку привязок HTTP и HTTPS для интерфейса узлов RESTful API, определенного в конкретных конфигурационных файлах приложения. Точно также как и для `gen_servers`, во всех модулях обратного вызова супервизора должна быть директива `-behavior(supervisor)`. Модули запускаются при помощи функций `start` или `start_link`, в которых могут быть необязательные параметры `ServerName`, `CallBackModule` и `Argument`, передаваемые в функцию обратного вызова `init/1`.

Если взглянуть на несколько первых строк кода в модуле `riak_core_sup.erl`, то наряду с директивой поведения и макросом, о которых мы расскажем далее, мы обнаружим функцию `start_link/3`:

```
-module(riak_core_sup).
-behavior(supervisor).
%% API
-export([start_link/0]).
%% Supervisor callbacks
-export([init/1]).
-define(CHILD(I, Type), {I, {I, start_link, []}, permanent, 5000, Type, [I]}).
start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, []).
```

В результате запуска супервизора будет порожден новый процесс и в модуле обратного вызова `riak_core_sup.erl` будет вызвана функция обратного вызова `init/1`. `ServerName` является кортежем в формате `{local, Name}` или `{global, Name}`, где `Name` является зарегистрированным именем супервизора. В нашем примере, как зарегистрированное имя, так и модуль обратного вызова являются атомом `riak_core_sup`, происходящим от макроса `?MODULE`. Мы в качестве аргумента передаем в `init/1` пустой список, который трактуется как значение `null`. Функция `init` является единственной функцией обратного вызова супервизора. Она должна возвращать кортеж следующего формата:

```
{ok, {SupervisorSpecification, ChildSpecificationList}}
```

где `SupervisorSpecification` является трехэлементным кортежем `{RestartStrategy, AllowedRestarts, MaxSeconds}`, содержащим информацию о том, что делать в случае разрушения или перезапуска процесса. `RestartStrategy` является одним из трех конфигурационных параметров, определяющих какое влияние должно оказываться на потомков при аварийном завершении поведения:

- `one_for_one`: влияние на другие процессы в дереве мониторинга не происходит.
- `rest_for_one`: процессы, запущенные после завершения процесса, завершаются и перезапускаются.
- `one_for_all`: все процессы завершаются и перезапускаются.

`AllowedRestarts` указывает, сколько раз любой из потомков супервизора может завершиться в течение `MaxSeconds` секунд прежде, чем будет завершен сам супервизор (и его потомок). Когда происходит завершение, в супервизор посыпается сигнал выхода EXIT, который, в соответствие с используемой стратегией перезапуска, обрабатывается определенным образом. Завершение супервизора после того, как будет достигнуто максимально допустимое количество перезагрузок, гарантирует, что не произойдет увеличение количества циклических перезагрузок и не возникнут другие проблемы, которые нельзя решить на этом уровне. Скорее всего, эта проблема, возникшая в процессе, будет локализована в отдельном поддереве, т. е. супервизор, получивший сообщение о распространении проблемы, завершит выполнение дерева, на которое проблема уже распространилась, и перезапустит его заново.

Если взглянуть на последнюю строку функции обратного вызова `init/1` в модуле `riak_core_sup.erl`, то мы увидим, что в этом конкретном супервизоре используется стратегия `one-for-one`, означающая, что процессы не зависят друг от друга. Супервизор разрешит максимум десять раз выполнить перезапуск прежде, чем он сам будет перезапущен.

В списке `ChildSpecificationList` определяется, какие потомки должны запускаться и отслеживаться супервизором, а также указывается информация, как их завершать и перезапускать. Он представляет собой список кортежей следующего формата:

```
{Id, {Module, Function, Arguments}, Restart, Shutdown, Type, ModuleList}
```

`Id` является уникальным идентификатором для этого конкретного супервизора. `{Module, Function, Arguments}` является экспортаемой функцией, результаты работы которой будут использованы в поведении при вызове функции `start_link` и будет возвращен кортеж вида `{ok, Pid}`. В стратегии определяется, что в зависимости от того, как завершится процесс, должно происходить, а именно:

- временный процесс типа `transient`, который никогда не перезапускается;
- временный процесс типа `temporary`, который перезапускается только в случае, если он завершился аварийно, и
- постоянный процесс типа `permanent`, который всегда перезапускается независимо от того, было ли его завершение нормальным или аварийным.

`Shutdown` является значением, указываемым в миллисекундах, которое используется в качестве времени, в течение которого поведению разрешается выполнять функцию `terminate` при ее перезапуске или при остановке системы. Можно также использовать атом бесконечного выполнения `infinity`, но это делать настоятельно не рекомендуется для поведений, не являющихся поведениями супервизора. `Type` является либо атомом рабочего процесса `worker`, если ссылка делается на общие серверы, обработчики событий и конечные автоматы, либо атомом `supervisor`. Вместе с `ModuleList`, списком модулей, реализующих поведение, они используются для управления процессами и их приостановкой во время выполнения процедуры обновления программного обеспечения.

чения. В списке спецификаций потомков можно указывать только поведения, которые уже существуют или реализованы пользователями и, следовательно, уже включены в дерево мониторинга.

Обладая этими знаниями, мы теперь сможем сформулировать стратегию перезапуска, в которой определены межпроцессные зависимости, пороги отказоустойчивости и ограничения эскалации работы процедур, базирующуюся на общей архитектуре. Мы также теперь должны суметь понять, что происходит в примере `init/1` модуля `riak_core_sup.erl`. Прежде всего, изучим макрос `CHILD`. Он создает спецификацию потомка для одного потомка, используя для этого имя модуля обратного вызова, например, `Id`, делая его постоянно используемым и задавая для него время завершения, равное 5 секундам. Потомки могут быть различного типа — рабочие процессы и супервизоры. Давайте взглянем на пример и посмотрим, что из него можно выяснить:

```
-define(CHILD(Id, Type), {Id, {Id, start_link, []}, permanent, 5000, Type, [Id]}).

init([]) ->
    RiakWebs = case lists:flatten(riak_core_web:bindings(http),
                                    riak_core_web:bindings(https)) of
        [] ->
            %% check for old settings, in case app.config
            %% was not updated
            riak_core_web:old_binding();
        Binding ->
            Binding
    end,

    Children =
        [?CHILD(riak_core vnode_sup, supervisor),
         ?CHILD(riak_core handoff_manager, worker),
         ?CHILD(riak_core handoff_listener, worker),
         ?CHILD(riak_core ring_events, worker),
         ?CHILD(riak_core ring_manager, worker),
         ?CHILD(riak_core node watcher_events, worker),
         ?CHILD(riak_core node watcher, worker),
         ?CHILD(riak_core gossip, worker) |
          RiakWebs
        ],
    {ok, {{one_for_one, 10, 10}, Children}}.
```

Большинство потомков `Children`, запущенных этим супервизором, являются статически заданными рабочими процессами (или в случае `vnode_sup`, супервизором). Исключением является часть `RiakWebs`, которая определяется динамически в зависимости от части HTTP конфигурационного файла `Riak`.

За исключением библиотечных приложений, каждое приложение OTP, в том числе и те, что есть в `Riak`, будет иметь свое собственное дерево мониторинга. В `Riak` различные приложения верхнего уровня работают в узле Erlang, например, `riak_core` — алгоритмы для распределенных систем, `riak_kv` — семантики хранилищ вида «ключ/значение», `webmachine` — для HTTP, и многие другие. Мы показали расширенное дерево для `riak_core` с тем, чтобы продемонстрировать, как происходит многоуровневый мониторинг. Одним из многих преимуществ этой структуры является то, что когда данная подсистема может выйти из строя (из-за ошибки, проблемы со средой окружения или преднамеренного действия), то можно ограничиться завершением работы только для поддерева первого экземпляра.

Супервизор перезапустит необходимые процессы и система в целом затронута не будет. На практике мы видели как это работает в случае использования `Riak`. Пользователь может обнаружить, что произошло разрушение виртуального узла, что требует только перезапуска супервизора `riak_core vnode_sup`. Если такое разрушение находится под контролем, то супервизор `riak_core` перезапустит нужный супервизор и предотвратит распространение остановок процессов суперви-

зоров более высокого уровня. Такая изоляция отказов и механизм восстановления позволяет разработчикам Riak (и Erlang) достаточно просто создавать устойчивые системы.

Значение супервизорной модели была продемонстрирована, когда один крупный промышленный пользователь создал очень сложную среду эксплуатации с тем, чтобы выяснить, где каждая из нескольких систем баз данных будет разрушена. В этой среде были случайным образом сгенерированы огромные нагрузки, причем как по объему трафика, так и по наличию отказов. Каково было его смузжение, когда Riak просто не остановил работу даже в таких непростых условиях. Конечно, если бы он заглянул внутрь, то обнаружил многочисленные остановки процессов или подсистем, но каждый раз супервизоры приводили все в порядок и заново запускали процессы и подсистемы, приводя, тем самым, всю систему обратно в рабочее состояние.

15.5.2. Приложения

Поведение `application`, которое мы ввели ранее, используется для упаковки модулей и ресурсов Erlang в виде компонентов, допускающих многократное использование. В OTP есть два вида приложений. Наиболее распространенная форма, называемая нормальными приложениями, запускает дерево мониторинга и все соответствующие статические рабочие процессы. В библиотечных приложениях, таких как стандартная библиотека Standard Library, которые поставляются как часть дистрибутива Erlang, содержатся библиотечные модули, но в них не происходит запуск дерева мониторинга. Это не значит, что в коде не может быть деревьев процессов или деревьев мониторинга. Это просто означает, что такие приложения запускаются как часть дерева мониторинга, принадлежащего другому приложению.

Система Erlang должна состоять из набора слабо связанных приложений. Некоторые из них пишут разработчики, некоторые из них доступны как проекты с открытым исходным кодом, а другие - являются частью дистрибутива Erlang/OTP. Система времени исполнения Erlang и его инструментальные средства работают со всеми приложениями одинаково независимо от того, являются ли те частью дистрибутива Erlang или нет.

15.6. Репликация и коммуникация в Riak

Riak был разработан для обеспечения высокой надежности и доступности при массовых нагрузках, причем на его разработку оказала влияние система хранения данных Dynamo компании Amazon [DHJ +07]. В архитектуре систем Dynamo and Riak объединены вместе особенности как распределенных хэш-таблиц DHT (Distributed Hash Tables), так и традиционных баз данных. Двумя ключевыми технологиями, которые используются в Riak и Dynamo, являются *последовательное хеширование (consistent hashing)*, применяемое для размещения реплики, и *протокол сплетен (gossip protocol)*, используемое для совместного доступа к общему состоянию.

Последовательное хеширование требует, чтобы все узлы системы знали друг о друге, и знали, каким разделом системы владеет каждый узел. Такие данные о назначениях можно хранить в централизованно управляемом конфигурационном файле, но для больших конфигураций это делать становится чрезвычайно трудно. Другой альтернативой является использование центрального конфигурационного сервера, но в результате в системе появляется место, из-за проблем в котором может произойти выход из строя всей системы. Вместо этого в Riak используется протокол сплетен, с помощью которого данные о вхождении узлов в кластер и принадлежности разделов системы распространяются по всей системе.

Протоколы сплетен, которые также называются протоколами эпидемий (epidemic protocols), работают именно так, как они называются. Когда узел в системе желает изменить часть совместно используемых данных, он делает изменения в своей локальной копии данных и сообщает об этом (сплетничает) другому узлу — своему случайному собеседнику. После получения обновления,

узел объединяет полученные изменения со своим собственным локальным состоянием и еще раз сплетничает с другим случайным собеседником.

Когда кластер Riak запускается, все узлы должны быть сконфигурированы так, что в каждом из них находится одинаковое количество разделов. Затем кольцо последовательного хеширования делится на количество разделов, и каждый интервал запоминается в виде пары `{HashRange, Owner}` (хэш-кольцо, владелец). Первый узел в кластере просто объявляет об этом всем разделам. Когда в кластер добавляется новый узел, он в существующем узле добавляется в список пар `{HashRange, Owner}` этого узла. Затем объявляется о парах (количество разделов)/(число узлов), обновляется локальное состояние, которое будет отражать новое распределение разделов. Затем информация о распределении разделов будет с помощью протокола сплетен передана другому узлу. А затем это обновленное состояние будет с помощью описанного ранее алгоритма распространено по всему кластеру.

При использовании протокола сплетен в Riak не появляется единое место отказа в виде централизованного конфигурационного сервера, что избавляет системных операторов от необходимости хранить критически важные конфигурационные данные о кластере. В этом случае любой узел может в системе маршрутизации запросов использовать данные о назначении разделов, полученные с помощью протокола сплетен. Совместное использование протокола сплетен и последовательного хеширования позволяет системе Riak действительно функционировать как децентрализованная система, что очень важно при развертывании и эксплуатации крупномасштабных систем.

15.7. Заключение и усвоенные уроки

Большинство программистов считают, что чем меньше и проще код, его не только проще поддерживать, но в нем часто также меньше ошибок. Благодаря базовым примитивам языка Erlang, имеющимся в дистрибутиве, разработку Riak стало возможным начинать со считающимся фундаментальным слоя асинхронных сообщений и создавать свои собственные протоколы, не беспокоясь о том, что лежит в основе их реализации. Когда Riak перерос в зрелую систему, в некоторых частях его сетевой коммуникации отказались от использования встроенных средств языка Erlang (и перешли к прямому манипулированию сокетами TCP), а в других - продолжают использовать хорошо подходящие для этих целей встроенные примитивы языка. Начав с использования нативных сообщения языка Erlang, позволяющих передавать любые сообщения, команда разработчиков Riak смогла очень быстро построить всю систему. Эти примитивы достаточно понятны и ясны, так что позже их было просто заменить в нескольких местах, где они не лучшим образом вписались в систему.

Кроме того, благодаря особенностям механизма передачи сообщений Erlang и легковесности ядра виртуальной машины Erlang, пользователь может одинаково легко запустить 12 узлов на 1 машине или 12 узлов на 12 машинах. Это делает существенно упрощает разработку и тестирование в сравнении с более тяжеловесными механизмами передачи сообщений и кластеризации. Это особенно ценно из-за принципиально распределенного характера системы Riak. Исторически известно, что в большинстве распределенных систем очень трудно работать в "режиме разработки" на ноутбуке одного разработчика. В результате, разработчики часто заканчивают тестирование своего кода в среде, которая является лишь частью полной системой и ведет себя не так, как полная система. Поскольку кластер Riak со множеством узлов может абсолютно просто работать на одном ноутбуке без чрезмерного потребления ресурсов или сложных трюков с конфигурацией, в процессе разработки можно достаточно легко создавать код, который будет пригоден для развертывания в промышленных условиях.

Использование супервизоров Erlang/OTP делает Riak гораздо более устойчивым в условиях выхода из строя подкомпонентов. Riak позволяет делать даже больше; благодаря такому поведению, кластер Riak также может достаточно легко поддерживать функционирование даже тогда, когда во всех узлах произошел сбой и они исчезли из системы. Это порой может привести к удивительному

уровню устойчивости. Одним из примеров этого был случай, когда крупное предприятие провело стресс-тестирование различных баз данных и умышленное их разрушение с целью выяснить их граничные возможности. Когда они добрались до Riak, у них возникли проблемы. Каждый раз, когда они находили способ (с помощью манипуляций на уровне операционной системы, плохих соединений IPC и т.д.), который должен был разрушить подсистему Riak, они могли наблюдать только очень короткий провал производительности, а затем система возвращалась к нормальному поведению. Это прямой результат вдумчивого подхода «пускай выходит из строя». Riak, когда это требовалось, перезапускал заново каждую из этих подсистем, а система в целом просто продолжала функционировать. Этот опыт показал, какой именно вид устойчивости Erlang/OTP вносит в создаваемые программы.

15.7.1. Благодарности

Эта глава основана на конспекте лекций 2009 года Франческо Чезарини (Francesco Cesarini) и Саймона Томпсона (Simon Thompson) Центрально-европейской школы функционального программирования, которая проводилась в Будапеште и Комарно. Основной вклад внес Саймон Томпсон (Simon Thompson) из Университета Кент в Кентербери, Великобритания. Отдельное спасибо всем рецензентам, которые высказывали свое мнение на различных этапах написания этой главы.

16. Проект Selenium WebDriver

Selenium является инструментом автоматизации браузера, обычно используемым для написания сквозных тестов веб-приложений. Инструмент автоматизации браузера делает именно то, что вы могли бы от него ожидать: автоматизирует управление браузером так, чтобы можно было автоматизировать повторяющиеся задачи. На первый взгляд эта проблема кажется простой, но, как мы увидим, для того, чтобы ее решить, за кулисами должно произойти многое.

Прежде, чем описывать архитектуру Selenium, полезно понять, как различные части проекта соотносятся друг с другом. На очень высоком уровне Selenium представляет собой набор из трех инструментов. Первый из этих инструментов, Selenium IDE, это расширение для Firefox, которое позволяет пользователям писать и выполнять тесты. Парадигмы написания/выполнения тестов может для некоторых пользователей оказаться недостаточно, поэтому в наборе есть второй инструмент, Selenium WebDriver, предоставляющий интерфейсы API на различных языках, которые предназначены для реализации большего контроля и применения стандартных приемов разработки программ. Последний инструмент, Selenium Grid, позволяет использовать интерфейсы Selenium API для управления отдельными экземплярами браузеров, работающими на разных машинах, образующих сетку, что дает возможность параллельно запускать большее количество тестов. В проекте, они называются - «IDE», «WebDriver» и «Grid». В этой главе рассматривается архитектура Selenium WebDriver.

Эта глава была написана в конце 2010 года, когда была реализована бета-версия Selenium 2.0. Если вы читаете эту книгу позже, то проект продвинется дальше и вы сможете увидеть, как были реализованы описанные здесь архитектурные решения. Если вы читаете ее раньше: Поздравляем! У вас есть машина времени. Можете ли вы сказать мне несколько выигрышных номеров лотереи?

16.1. История

Джейсон Хогинс (Jason Huggins) приступил к проекту Selenium в 2004 году, когда он работал в ThoughtWorks над внутренним проектом компании — системой Time and Expenses (T&E), в которой широко использовался Javascript. Хотя в то время доминирующим браузером был Internet Explorer, в ThoughtWorks использовали ряд альтернативных браузеров (в частности, варианты Mozilla) и в случаях, когда приложение T&E не работало с выбранным браузером, требовалось со-

бирать сообщения об ошибках. Инструменты тестирования с открытым исходным кодом, которые были на тот момент, либо предназначались только для одного браузера (обычно - для IE), либо моделировали работу браузера (например, HttpUnit). Стоимость лицензии на коммерческий инструмент могла оказаться разорительной для ограниченного бюджета небольшого внутреннего проекта, поэтому это даже не рассматривались в качестве допустимого варианта решения.

Там, где трудно автоматизировать, обычно полагаются на ручное тестирование. Такой подход не годится в ситуациях, когда команда разработчиков очень маленькая или когда релизы выходят очень часто. Также зря растратываются человеческие ресурсы, когда просят использовать скрипт, который может быть автоматизирован. Более прозаично то, люди медленнее, чем машина, выполняют скучные повторяющиеся задачи и оно при этом более подвержены ошибкам. Ручное тестирование не могло быть вариантом решения.

К счастью, во всех браузерах, в которых выполняется тестирование, поддерживают язык Javascript. Это привело Джейсона и его команду к мысли написать на этом языке инструмент тестирования, который можно было бы использовать для проверки поведения приложения. Вдохновленные работой, которая была проведена в рамках проекта FIT [1], они создали поверх языка Javascript синтаксические конструкции, имеющие вид таблиц, которые позволили тем, кто имел мало опыта в программировании, писать в файлах HTML тесты, в которых использовались ключи. Этот инструмент, первоначально называвшейся «Selenium», но позже переименованный в «Selenium Core», был выпущен в 2004 году под лицензией Apache 2.

Табличный формат в Selenium аналогичен по структуре формату ActionFixture из FIT. Каждая строка таблицы разделена на три столбца. В первом столбце указывается имя команды, которая должна быть выполнена, во втором столбце обычно находится идентификатор элемента, а в третьем столбце указывается дополнительное значение. Например, присваивание строки «Selenium WebDriver» элементу, идентифицируемому именем «q», осуществляется следующим образом:

```
type      name=q      Selenium WebDriver
```

Поскольку Selenium был написан на чистом языке Javascript, его первоначальный вариант для того, чтобы не нарушать правил политики безопасности браузера и не выходить за границы песочницы Javascript, требовал от разработчиков размещать пакет Core и его тесты на том же самом сервере, что и тестируемое приложение AUT (the application under test). Это не всегда было удобно или возможно. Еще хуже было то, что хотя оболочка IDE, имеющаяся у разработчиков, предоставляла им возможность оперативно управлять кодом и ориентироваться в коде большого размера, для HTML такого инструмента не было. Быстро стало ясно, что поддержка наборов тестов даже средних размеров, громоздко и болезненно [2].

Для решения этого и других вопросов был написан HTTP прокси-сервер, который в Selenium мог перехватывать каждый запрос HTTP. Использование такого прокси-сервера позволило обойти многие из ограничений политики «только одного хоста», когда браузер не позволяет языку Javascript обращаться куда либо, кроме сервера, с которого была взята текущая страница, и это позволило смягчить первые недостатки. Такая схема дала возможность написать привязки Selenium для нескольких языков: в них было просто необходимо иметь возможность отправлять запросы HTTP на конкретный URL. Этот сетевой формат был тщательно промоделирован с помощью табличного синтаксиса Selenium Core, и стал, вместе с табличным синтаксисом, называться «Selenese». Поскольку языковыми привязками можно было управлять из браузеров на расстоянии, этот инструмент был назван «Selenium Remote Control» (механизмом дистанционного управления Selenium) или «Selenium RC».

Когда разрабатывался проект Selenium, в ThoughtWorks зарождался еще один фреймворк автоматизации браузеров: WebDriver. Исходный код для него был создан в начале 2007 года. WebDriver был создан в результате работы над проектами, в которых было необходимо изолировать всю линейку тестов, выполнявшихся на проектами так, чтобы используемые для тестов инструменты не

влияли на проекты. Как правило, подобная изоляция выполняется с помощью шаблона Adapter. WebDriver появился в результате практического опыта, полученного при последовательном применении данного подхода в многочисленных проектах, и первоначально он представлял собой обертку вокруг HtmlUnit. Вскоре после его выпуска последовала поддержка для Internet Explorer и Firefox.

Когда WebDriver был реализован, он существенно отличался от Selenium RC, хотя оба пакета относились к одной и той же нише интерфейсов API, предназначенных для автоматизации браузеров. Самым очевидным различием для пользователя было то, что в Selenium RC был интерфейс API, в котором использовался словарь и в котором все методы были доступны из одного класса, тогда как в WebDriver был более объектно-ориентированный интерфейс API. Кроме того, в WebDriver поддерживалась работа только с языком Java, в то время как в Selenium RC предлагалась поддержка широкого спектра языков. Были также существенные технические различия: пакет Selenium Core (на основе которого был создан RC) был, по существу, приложением на JavaScript, работающим внутри изолированной среды браузера. В WebDriver за счет существенных затрат, вложенных в разработку самого фреймворка, была сделана попытка связать его с самим браузером с тем, чтобы можно было обойти модель безопасности, используемую в браузере.

В августе 2009 года было объявлено, что эти два проекта будут объединяться, и результатом этого объединения стал проект Selenium WebDriver. На момент написания данной главы, в WebDriver поддерживается привязка к таким языкам, как Java, C#, Python и Ruby. В нем предлагается поддержка для браузеров Chrome, Firefox, Internet Explorer, Opera и браузеров, используемых в Android и в iPhone. Есть дочерние проекты, которые не хранятся в этом репозитории исходного кода, но тесно взаимосвязаны с основным проектом и обеспечивают привязку к языку Perl, реализацию для браузера в BlackBerry и для "безоконного варианта" WebKit, что удобно в тех случаях, когда тесты нужно запускать с постоянным взаимодействием с сервером, но отображение не требуется. Продолжается поддержка исходного механизма Selenium RC и обеспечивается поддержка WebDriver для тех браузеров, работа с которыми по иному не поддерживается.

16.2. Пару слов о жаргоне

К сожалению, в проекте Selenium много жаргона. Напомним, с чем мы уже сталкивались:

- *Selenium Core* является сердцем первоначальной реализации Selenium и представляет собой набор скриптов Javascript, которые управляют браузером. Их иногда называют «Selenium», а иногда – как «Core».
- *Selenium RC* является названием, которое было дано языковым привязкам в Selenium Core, которые обычно и несколько неправильно называют просто как «Selenium» или «RC». Оно было заменено на Selenium WebDriver, причем интерфейс API из RC называют «Selenium 1.x API».
- *Selenium WebDriver* вписывается в ту же нишу, что и RC, и включает в себя исходные привязки 1.x. Это относится как к языковым привязкам, так и к реализации кода, управляющего отдельными браузерами. Его обычно называют просто как «WebDriver» или иногда как Selenium 2. Несомненно, что с течением времени он войдет в состав «Selenium».

Внимательный читатель заметил, что термин «Selenium» используется в довольно широком смысле. К счастью, обычно из контекста становится ясно, какой именно «Selenium» имеется в виду.

Наконец, есть еще один термин, которым я буду пользоваться, и я представлю его достаточно просто: «драйвер» это название, которое дается конкретной реализации WebDriver API. Например, есть драйвер Firefox и драйвер Internet Explorer.

16.3. Вопросы архитектуры

Прежде чем мы начнем разбираться с отдельными частями с тем, чтобы понять, как они связаны друг с другом, полезно разобраться с общими архитектурными вопросами и вопросами разработки проекта. Кратко говоря, это следующие вопросы:

- Снижение расходов.
- Имитация поведения пользователя.
- Перекладывание всей работы на драйвера ...
- ... и вам не потребуется разбираться с тем, как это все работает.
- Уменьшение влияния фактора автобуса.
- Следует дружественно относиться к реализации языка Javascript.
- Вызов каждого метода является дистанционным вызовом RPC.
- Это проект с открытым исходным кодом.

16.3.1. Снижение расходов

Поддержка для X браузеров на Y платформах изначально дорогостоящее занятие, как с точки зрения первоначальной разработки, так и сточки зрения сопровождения. Если мы сможем найти какой-нибудь способ сохранять высокое качество продукта, не нарушив при этом слишком много других принципов, то это именно то, что нам могло бы подойти. Наиболее ярко это видно по нашему выбору везде, где это возможно, использовать язык Javascript, о чём вы скоро прочитаете.

16.3.2. Эмуляция поведения пользователя

WebDriver предназначен для точной имитации того, как пользователь будет взаимодействовать с веб-приложением. Обычным подходом в имитации пользовательского ввода является применение языка Javascript для создания ряда событий, которые приложение будет воспринимать, как если бы реальный пользователь выполнял те же самые действия. Этот подход с использованием «синтезированные события» чреват трудностями, поскольку каждый браузер, а иногда и разные версии одного и того же браузера, ведут себя немного по-разному. Ситуация усложняется тем, что в большинстве браузеров пользователю по соображениям безопасности не разрешено использовать такие элементы форм, как ввод файлов.

Везде, где это возможно, WebDriver использует альтернативный подход запуска событий на уровне операционной системы. Поскольку такие «нативные события» нельзя генерировать браузером, такой подход позволяет обойти ограничения безопасности, накладываемые на генерируемые события, а поскольку они характерны для конкретной ОС, то раз уж они используются для одного браузера на конкретной платформе, то их сравнительно легко использовать и для другого браузера. К сожалению, такой подход является единственным возможным, когда WebDriver может тесно взаимодействовать с браузером и когда команда разработчиков определяет, как лучше посыпать нативные события и, при этом, не передавать фокус окну браузеру (тесты Selenium выполняются в течение долгого времени и, пока они работают, было бы хорошо иметь возможность использовать машину для выполнения других задач). На момент написания данной главы это означало, что нативные события можно было использовать в Linux и в Windows, но не в Mac OS X.

Независимо от того, каким образом WebDriver эмулирует пользовательский ввод, мы стараемся настолько точно имитировать поведение пользователя, насколько это возможно. Это контрастирует с использованием RC, в котором содержится API, действующее на гораздо более низком уровне, чем работает пользователь.

16.3.3. Перекладывание всей работы на драйвера

Может быть, это прозвучит банально, как, например, то, что вода должна быть мокрой, но я считаю, что нет смысла писать код, если он не работает. То, что драйверы работают в проекте Selenium мы доказываем тем, что у нас есть достаточно много наборов автоматизированных тестов. Они, как правило, являются «интеграционными тестами», требующими компиляции кода и

используемыми при взаимодействии браузера с веб-сервером, но там, где это возможно, мы пишем «юнит-тесты», которые, в отличие от интеграционных тестов, могут работать без полной перекомпиляции. На момент написания главы было приблизительно 500 интеграционных тестов и приблизительно 250 юнит-тестов, которые можно было выполнять с каждым браузером. Сейчас их больше, поскольку мы выпускаем новые версии и пишем новый код и наш интерес смещается в сторону написания юнит-тестов.

Не каждый тест работает с каждым браузером. Некоторые проверяют специфические возможности, которые не поддерживаются в некоторых браузерах или которые в разных браузерах обрабатываются по-разному. Примерами могут служить тесты для новых возможностей HTML5, которые поддерживаются не всеми браузерами. Несмотря на это, для каждого из основных браузеров, работающих на настольных машинах, есть достаточно важно подмножество тестов, работающих именно с ним. Понятно, что найти способ, как для каждого браузера на нескольких платформах запустить более 500 тестов, является серьезной задачей и это одна из тех задач, с решением которых продолжает бороться проект.

16.3.4. Вам не нужно понимать, как это все работает

Очень немногие разработчики достаточно опытные, что им комфортно работать с всеми языками и технологиями, которыми мы пользуемся. Поэтому наша архитектура должна позволить разработчикам сосредоточить свои таланты на том, где они могут принести наибольшую пользу, и не требовать от них работать с теми частями кода, с которыми им работать некомфортно.

16.3.5. Уменьшение влияния фактора автобуса

Есть (не очень серьезная) концепция разработки программ, называемая «фактор автобуса». Она касается ряда ключевых разработчиков, с которыми может произойти нечто ужасным — предположим, их скажет автобус — в результате проект может быть оставлен в том состоянии, в котором его не удастся продолжить. Некоторые такие сложные проекты, как автоматизация браузеров, особенно подвержены влиянию этого фактора, поэтому в многих наших архитектурных решениях этот фактор учитывается как можно ответственнее.

16.3.6. Следует дружественно относиться к реализации языка Javascript

WebDriver в случае, если нет другого способа управлять браузером, возвращается обратно к управлению с помощью чистого языка Javascript. Это означает, что любой интерфейс API, который мы добавляем, должен быть дружественным к реализации языка Javascript. Что касается конкретного примера, то в HTML5 добавлено локальное хранилище LocalStorage — интерфейс API для хранения структурированных данных на клиентской стороне. Это обычно реализовано в браузере с использованием пакета SQLite. Естественной реализацией было бы реализовать подключение к базе данных с помощью находящегося ниже механизма хранения данных, использующего что-нибудь вроде JDBC. В конце концов, мы остановились на интерфейсе API, который достаточно точно моделирует лежащую ниже реализацию Javascript, т. к. все, что моделирует обычные интерфейсы API, используемые для доступа к базам данных, не слишком дружественно к реализациям языка Javascript.

16.3.7. Каждый вызов — это дистанционный вызов RPC

WebDriver осуществляет управление браузерами, которые работают в других процессах. Хотя проще это не учитывать, но это означает, что каждый вызов, который сделан через интерфейс API фреймворка, является дистанционным вызовом и, следовательно, производительность фреймворка всецело зависит от задержек в сети. Т.к. в большинстве ОС для localhost маршрутизация оптимизирована, то в обычном режиме работы, это, быть может, не очень заметно, но как только сетевые

задержки между браузером и кодом теста становятся больше, для разработчиков интерфейса API и его пользователей начинает казаться, что эффективность уменьшается.

Это оказывает определенное влияние на разработку API. В большом API с грубо сделанными функциями можно было бы сократить задержку за счет отказа от использования множественных вызовов, но при этом следует соблюсти баланс между выразительными возможностями и простотой использования API. Например, есть несколько проверок, которые нужно сделать для того, чтобы определить, является ли элемент видимым для конечного пользователя. Мы должны принять во внимание не только различные настройки CSS, что, подразумевается, может потребовать просмотра родительских элементов, но нам, вероятно, также следует проверить размеры элемента. В минималистском варианте интерфейса API необходимо, чтобы каждая из этих проверок делалась отдельно. В WebDriver они объединены в одном методе `isDisplayed`.

16.3.8. Заключительный штрих: Это проект с открытым исходным кодом

Хотя это не совсем архитектурный аспект, но Selenium является проектом с открытым исходным кодом. Основная мысль, которая объединяет все вышеуказанные аспекты вместе, в том, что мы хотели сделать так, чтобы каждому новому разработчику было бы проще вносить в проект свой собственный вклад. Мы надеемся добиться этого за счет того, что поддерживаем минимальные требования к уровню знаний, необходимых новым разработчикам, используем минимально необходимое количество языков и для того, чтобы проверить, что ничего не испорчено, используем автоматизированные тесты.

Первоначально проект был поделен на ряд модулей, причем каждый модуль, представляющий конкретный браузер, имел дополнительные модули, реализующий обычный код, и модули с кодами поддержки и утилит. В этих модулях также хранились деревья исходных кодов для привязок к конкретным языкам. Этот подход был достаточно осмысленным для таких языков, как Java и C#, но создавал проблемы для программистов, работающих с языками Ruby и Python. Это почти сразу привело к тому, что интерес к проекту проявило только очень немного пользователей, способных и заинтересованных использовать привязки к языкам Python и Ruby. Чтобы решить эту проблему, в октябре и ноябре 2010 года исходный код был реорганизован так, что для языков Ruby и Python код хранился в папках верхнего уровня отдельных для каждого языка. Это больше соответствовало ожиданиям разработчиков открытого исходного кода, использующих эти языки, и практически сразу это дало эффект в повышении вклада, делаемого сообществом разработчиком.

16.4. Побеждаем сложность

Программа является конструкцией, состоящей из отдельных частей. Отдельные части являются сложными, и мы, как разработчики интерфейса API, должны выбирать, как справляться с этой сложностью. С одной стороны, мы могли бы как можно равномернее распределить эту сложность по всему интерфейсу, а это означало бы, что с этой сложностью столкнется каждый, кто пользуется интерфейсом API. Другая крайность предполагает все, что связано со сложностью, максимально собрать в одном месте. Это единственное место будет местом тьмы и ужаса для тех, кто рискнет им воспользоваться, но компромисс будет в том, что для пользователей API, которым не нужно вникать в реализацию, проблема сложности будет заранее решена.

Разработчики WebDriver больше склоняются к поиску и изоляции сложности в небольшом количестве мест, а не распределению ее по всему проекту. Одна из причин этого связана с нашими пользователями. Как видно по списку наших ошибок, пользователи исключительно хорошо находят проблемы и задают вопросы, но поскольку многие из них не являются разработчиками, сложный интерфейс API не будет способствовать хорошей работе. Мы стремились разрабатывать интерфейс API, который помогает людям двигаться в правильном направлении. В качестве примера, рассмотрим следующие методы из исходного интерфейса Selenium API, каждым из которых можно пользоваться при установке значений вводимых элементов:

- type
- typeKeys
- typeKeysNative
- keydown
- keypress
- keyup
- keydownNative
- keypressNative
- keyupNative
- attachFile

В WebDriver API их эквивалент следующий:

- sendKeys

Как уже ранее обсуждалось, здесь отражено одно из основных философских различий между RC и WebDriver, состоящее в том, что WebDriver стремится подражать пользователям, тогда как в RC предлагаются интерфейсы API, функционирующие на более низком уровне, к которым, по мнению пользователя, сложно или невозможно получить доступ. Различие между методами typeKeys и typeKeysNative в том, что в первом случае всегда используются синтезируемые события, а во втором для ввода значений нажатых клавиш делаются попытки использовать AWT Robot. К сожалению, AWT Robot посылает информацию о нажатии клавиш в окно, имеющее фокус, которое не обязательно может быть браузером. Нативные события в WebDriver, напротив, направляются непосредственно в дескриптор окна, что позволяет не требовать от окна браузера, чтобы оно имело фокус.

16.4.1. Схема WebDriver

Команда разработчиков считает интерфейс WebDriver API «объектно-ориентированным². Интерфейсы строго определены и мы стараемся, чтобы у каждого из них была только одна роль или функция, но вместе того, чтобы моделировать каждой отдельно взятый тег HTML в виде отдельного класса, у нас есть только один интерфейс WebElement. В соответствие с этим подходом, разработчики, которые используют IDE, поддерживающую автозавершение, могут переходить к выполнению следующего шага. В результате этого, кодирование может выглядеть (для языка Java) следующим образом:

```
WebDriver driver = new FirefoxDriver();
driver.<пользователь нажимает на пробел>
```

В этот момент появляется относительно короткий список из 13 методов, из которых нужно сделать выбор. Пользователь выбирает один из них:

```
driver.findElement(<пользователь нажимает на пробел>)
```

В большинстве IDE в это момент выпадает подсказка о типе ожидаемого аргумента, в нашем случае - объекты By. Для объектов By есть целый ряд предварительно сконфигурированных методов factory, которые для самого объекта By объявлены как статические методы. Наш пользователь быстро завершит ввод строки кода, которая будет выглядеть следующим образом:

```
driver.findElement(By.id("некоторое_id"));
```

Ролевые интерфейсы

Рассмотрим упрощенный класс Shop (Магазин). Каждый день в магазин должны поступать товары и, поэтому, он взаимодействует с классом Stockist (Поставщик), который нужен для поставки но-

вой партии товара. Каждый месяц нужно платить сотрудникам и оплачивать налоги. Давайте предположим, что это делается с помощью класса Accountant (Бухгалтер). Один из способов моделирования этой ситуации выглядит следующим образом:

```
public interface Shop {
    void addStock(StockItem item, int quantity);
    Money getSalesTotal(Date startDate, Date endDate);
}
```

У нас есть два вспомогательных класса, как задать границу при определении интерфейса между магазином Shop, бухгалтером Accountant и поставщиком Stockist. Теоретически мы могли бы провести линию так, как показано на рис.16.1.

Это означает, что классы Accountant и Stockist должны восприниматься классом Shop как аргументы с их соответствующими методами. Однако недостатком здесь является то, что маловероятно, что бухгалтеру действительно нужны полки с товарами, и, вероятно, не лучшая идея реализовывать в классе Stockist огромные наценки на стоимость товаров, которые делаются в магазине. Таким образом, лучше нарисовать линию так, как это показано на рис.16.2.

Нам понадобится два интерфейса, которые нужно реализовать в классе Shop, но в этих интерфейсах точно определяется роль, которую выполняет класс Shop (Магазин) для обоих классов Accountant (Бухгалтер) и Stockist (Поставщик). Это следующие ролевые интерфейсы:

```
public interface HasBalance {
    Money getSalesTotal(Date startDate, Date endDate);
}

public interface Stockable {
    void addStock(StockItem item, int quantity);
}

public interface Shop extends HasBalance, Stockable {
```

Я считаю, что исключительные состояния UnsupportedOperationExceptions и иже с ними глубоко неприятны, но должно быть что-то, что позволяет использовать эти функции некоторой группе пользователей, которым это, возможно, нужно, и не загромождает остальные интерфейсы API для большинства пользователей. Для этого в WebDriver широко используется ролевой интерфейс. Например, есть интерфейс JavascriptExecutor, который в контексте текущей страницы позволяет выполнять произвольные куски кода на Javascript. Успешное приведение экземпляра WebDriver к этому интерфейсу указывает, что вы можете в работе использовать эти методы.

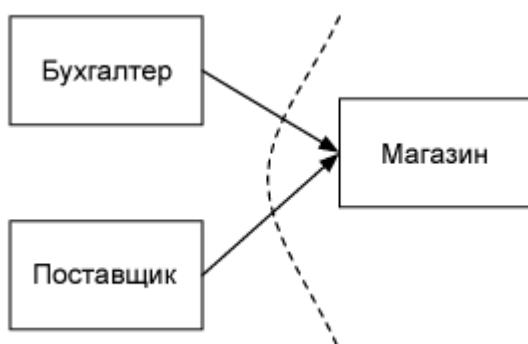


Рис.16.1: Классы Accountant (Бухгалтер) и Stockist (Поставщик) зависят от класса Shop (Магазин)

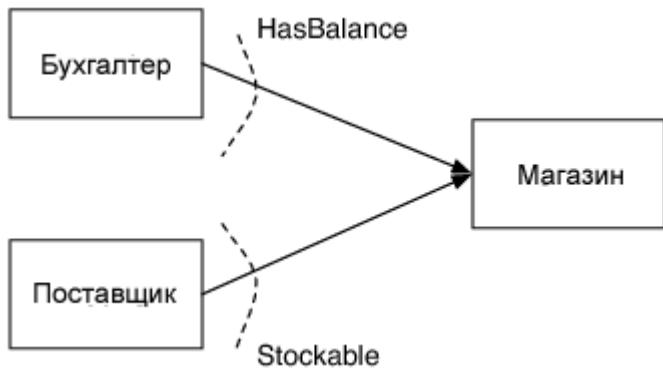


Рис.16.2: В классе Shop (Магазин) реализованы интерфейсы HasBalance (Получить баланс) и Stockable (Поступление товаров)

16.4.2. Комбинаторный взрыв

Первое, что сразу приходит на ум при мысли об огромном количестве браузеров и языков, которые поддерживает WebDriver, что если не принять меры, можно быстро столкнуться с нарастающими затратами его поддержки. При наличии X браузеров и Y языков очень легко попасть в ловушку, где требуется поддерживать $X \times Y$ реализаций.

Одним из способов уменьшения этих затрат могло бы быть уменьшение количества языков, которые поддерживает WebDriver, но мы не хотели идти по этому пути по двум причинам. Во-первых, когда переходишь с одного языка на другой, то это происходит за счет определенной когнитивной нагрузки, так что поскольку пользователей фреймворка могут писать свои тесты на том же самом языке, на котором они выполняют большую часть своей работы, у них имеется преимущество. Во-вторых, когда в одном проекте смешиваются несколько языков, команда разработчиков может чувствовать себя некомфортно и часто случается, что корпоративные стандарты кодирования и требования проектирования обязывают придерживаться технологического единства (хотя, я думаю, приятно то, что со временем эта вторая причина становится менее значимой), поэтому недопустимо уменьшать количество поддерживаемых языков.

Сокращение числа поддерживаемых браузеров так же не является вариантом — было много крика, когда мы прекратили в WebDriver поддержку Firefox 2, несмотря на то, что, когда мы так поступили, этот браузер занимал менее 1% рынка браузеров.

Единственное, что нам осталось выбрать, это сделать так, чтобы все браузеры с точки зрения привязки к языкам выглядели идентично: нужно предложить единый интерфейс, который мог бы подходить для самых различных языков. И еще мы хотели, чтобы привязки к языкам было бы настолько просто писать, насколько это возможно, что предполагало, что мы хотим сделать настолько тонкими, насколько это возможно. Чтобы поддержать такой подход, мы перенесли в драйвер, лежащий ниже, столько логики, сколько было возможным: то, что мы не могли перенести в драйвер, это те функциональные возможности, которые нужно реализовывать в каждом языке, который мы поддерживаем, и это может потребовать существенного объема работ.

Например, в драйвере IE функции, касающиеся местонахождения и запуска IE, были успешно перенесены в логику основного драйвера. Хотя это и удивило нас тем количеством строк кода, которое стало в драйвере, привязка к языку при создании нового экземпляра драйвера свелась к единственному вызову метода, находящегося в этом драйвере. Для сравнения, в драйвере Firefox такие изменения сделать не удалось. Только для языка Java это означает, что у нас есть три основных класса, занимающихся конфигурированием и запуском Firefox, размер которых равен приблизительно 1300 строкам кода. Эти классы дублируются для каждой привязки к языку, в котором нужно поддерживать FirefoxDriver и следует полагаться на запуск сервера Java. В результате это свело к сопровождению большого количества кода.

16.4.3. Недостатки схемы, используемой в WebDriver

Недостаток решения предоставлять так возможности драйвера заключается в том, что до тех пор, пока пользователи не знают, что существует конкретный интерфейс, они не смогут догадаться, что WebDriver поддерживает функциональные возможности такого рода: в API нет возможность самостоятельно узнать об этих возможностях. Конечно, когда WebDriver был новинкой, нам казалось, что мы потратим много времени, просто рассказывая о конкретных интерфейсах. Теперь мы тратим существенно больше усилий на нашу документацию и по мере того, как API используется все шире и шире, пользователям становится все легче и легче находить нужную им информацию.

В частности, есть место, где, по моему мнению, наш интерфейс API неудовлетворителен. У нас есть интерфейс, называемый `RenderedWebElement`, который представляет собой странное объединение методов, которые делают запросы обновленного состояния элемента (`isDisplayed`, `getSize` и `getLocation`), выполняют с ним операции (метод `hover` и метод объекта захвата и его перетаскивания), и удобного метода, с помощью которого можно значение, установленного определенному свойству CSS. Интерфейс был создана, потому что в драйвере `HtmlUnit` необходимая информация не предоставлялась, а в драйверах Firefox и IE - предоставлялась. Первоначально в нем был только первый набор методов, но прежде, чем я успел подумать о том, как я бы хотел, чтобы этот API выглядел, мы добавили в него другие методы. Сейчас этот интерфейс хорошо известен и трудно решить, сохранять ли этот неприглядный уголок API, учитывая то, что он широко используется, или попытаться его удалить. Я предпочитаю не оставлять за собой "мусор", так что прежде, чем мы выпустим Selenium 2.0, важно решить эту проблему. В результате, к тому времени, когда вы будете читать статью, интерфейс `RenderedWebElement`, возможно, будет уже исправлен.

С точки зрения разработчиков, жесткое подключение к браузеру, хотя оно и неизбежно, также является конструктивным недостатком. Приходится предпринимать значительные усилия для того, чтобы поддерживать новый браузер, и зачастую для того, чтобы осуществить это правильно, нужно сделать нескольких попыток. Что касается конкретного примера, то драйвер Chrome прошел через четыре полных этапа переписывания, а драйвер IE также прошел через три основных этапа переписывания. Преимущество жесткой привязки к браузеру состоит в предоставлении больших возможностей управления.

16.5. Уровни и Javascript

Инструментальные средства автоматизации браузера, в сущности, состоят из трех частей:

- Способы взаимодействия с DOM.
- A mechanism for executing Javascript. Механизма выполнения Javascript.
- Some means of emulating user input. Некоторых средств эмуляции пользовательского ввода.

Данный раздел посвящен первой части: способу взаимодействия с DOM. Универсальным языком браузера является язык Javascript, и похоже, что это идеальный язык для использования при взаимодействии с DOM. Хотя этот выбор и кажется очевидным, если его сделать, то это приведет к некоторым интересным проблемам и влияющим друг на друга требованиям, баланс которых, когда рассматривается Javascript, нужно соблюсти.

Как и в большинстве крупных проектов, в Selenium используется многоуровневый набор библиотек. Нижним уровнем является библиотека Closure Library от Google, в которой предоставляются примитивы и механизм модульности, благодаря которым обращение к исходным файлам сводится к минимуму. Поверх этого слоя расположена библиотека утилит, предоставляющая функции, варьирующиеся от простых задач, таких как получение значения атрибута или определения того, будет ли элемент виден конечному пользователю, до гораздо более сложных действий, таких как моделирование щелчка кнопки мыши с помощью синтезируемых событий. В рамках проекта они рассматриваются как наименьшие единицы браузерной автоматизации, и поэтому называются

атомами браузерной автоматизации Browser Automation Atoms или просто атомами. Наконец, есть уровни адаптеров, на которых атомы собираются в соответствие с контрактами как WebDriver, так и Core.

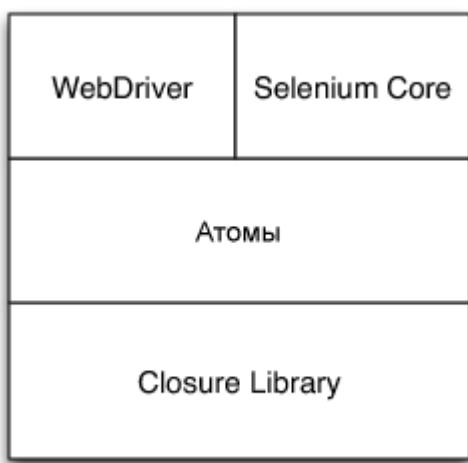


Рис. 16.3: Уровни библиотеки Javascript в Selenium

Библиотека Closure Library была выбрана по нескольким причинам. Главным было то, что механизм модульности, используемый в библиотеке, воспринималась компилятором Closure Compiler. Компилятор Closure Compiler - это компилятор, выходным языком которого является Javascript. "Компиляция" может быть простой, например, упорядочивание входных файлов в порядке их обращения к зависимостям, объединение их и выдачи их на печать в красивом виде, или может быть сложной, например, максимальное сжатие размеров файлов и удаление недостижимых кусков кода. Другим неоспоримым преимуществом было то, что некоторые члены команды, работающие с кодом на Javascript, были хорошо знакомы с библиотекой Closure Library.

Эта библиотека «атомов» кода используется везде во всем проекте, когда нужно обратиться к DOM. Для RC и тех драйверов, которые созданы, в основном, на Javascript, библиотека используется напрямую и компилируется, как правило, в виде монолитного скрипта. Для драйверов, написанных на Java, отдельные функции, находящиеся в слое адаптера WebDriver, компилируются с включенной полной оптимизацией, а генерированный Javascript добавляется в JAR архивы в виде ресурсов. Для драйверов, написанных на вариантах С, например, драйверов iPhone и IE, не только отдельные функции компилируются с полной оптимизацией, но и весь результат конвертируется в константу, указываемую в заголовке, которую можно по требованию выполнить с помощью обычного механизма выполнения Javascript, имеющегося в драйвере. Хотя все эти действия и кажутся странным, но они позволяют поместить Javascript в драйвер, лежащий ниже, и пропадает необходимость во многих местах добавлять соответствующий код.

Поскольку атомы используются везде, где можно, легко и быстро обеспечивается согласованное поведение различных браузеров, а поскольку библиотека написана на Javascript, ей для выполнения в цикле разработки не требуется повышенных привилегий. Библиотека Closure Library может загружать зависимости динамически, поэтому разработчику Selenium нужно только написать тест и загрузить его в браузер, по мере необходимости изменяя код и нажимая кнопку обновления. После того, как тест проходит в одном браузере, его можно легко загрузить в другой браузер и убедитесь, что он проходит и в нем. Поскольку в библиотеке Closure Library хорошо реализовано абстрагирование от различий в браузерах, часто достаточно знать, и это отрадно, что есть непрерывно создаваемые сборки, которые будут выполнять тестовый набор в каждом поддерживаемом браузере.

Первоначально в Core и WebDriver было много фрагментов совпадающего кода - кода, который выполнял одну и ту же функцию, но немного по-разному. Когда мы начинали работу над атомами, этот код был детально просмотрен с целью найти для этих функций "лучший вариант реализации".

В конце концов, оба проекта уже широко использовались и их код был очень надежен, так что было бы не только расточительно, но и глупо выбросить все и все начать с нуля. Когда были выделены атомы, были определены места, где каждый из них должен был использоваться, и мы перешли использование атомов. Например, в методе `getAttribute` драйвера Firefox число строк было сокращено примерно с 50 до 6, включающих пустые строки:

```
FirefoxDriver.prototype.getAttribute =
  function(respond, parameters) {
    var element = Utils.getElementAt(parameters.id,
      respond.session.getDocument());
    var attributeName = parameters.name;

    respond.value = webdriver.element.getAttribute(element, attributeName);
    respond.send();
};
```

Во второй от конца строке, где переменной `respond.value` присваивается значение, используется библиотека атомов WebDriver.

Атомы являются практической демонстрацией нескольких аспектов, касающихся архитектуры проекта. Естественно, соблюдается требование, что реализация API полностью соответствовала реализации языка Javascript. Еще лучше то, что одна и та же библиотека используется во всем коде; если одна и та же ошибка будет выявлена во многих реализациях, то теперь ее достаточно исправить в одном месте, что снижает затраты на изменение при одновременном повышении стабильности и эффективности. Атомы также делают в проекте более удобной работу с ошибками. Поскольку для того, чтобы проверить, что исправление работает, можно использовать обычные тестовые юниты Javascript, барьер для присоединения к работе над проектом с открытым кодом значительно ниже, чем в случае, когда нужно знать, как реализован каждый драйвер.

Есть еще одно преимущество использования атомов. Для команд разработчиков, которые хотят в контролируемом режиме перейти к самым новым интерфейсам WebDriver API, важным инструментом является слой, эмулирующий существующую реализацию RC, но поддержка которого осуществляется драйвером WebDriver. Поскольку Selenium Core атомизирован, можно компилировать каждую функцию по отдельности, что делает задачу написания этого эмулирующего слоя более простой в реализации и более аккуратной.

Само собой разумеется, что есть и отрицательные стороны подхода. Самое важное, что компиляция Javascript в константу const языка C выполняется очень странно и это всегда сбивает с толку новых участников проекта, которые хотят работать в коде на языке C. Также очень редко, у кого из разработчика есть все версии всех браузеров, и редко, кто запускает все тесты на всех этих браузерах — возможно, что для кого-то это может в неожиданном месте ненароком привести к проблеме и в частности в тех случаях, когда насылаются новые непрерывно создаваемые сборки, может потребоваться некоторое время для того, чтобы идентифицировать проблему.

Поскольку атомы нормализуют значения, возвращаемые браузерами, возвращаемые значения также могут оказаться неожиданными. Например, рассмотрим следующий фрагмент HTML:

```
<input name="example" checked>
```

Значение атрибута `checked` будет зависеть от используемого браузера. Атомы нормализуют его, а также другие логические атрибуты, определяемые в спецификации HTML5, так, чтобы они были равны «`true`» или «`false`». Когда этот атом был добавлен в код, мы обнаружили много мест, где принимались решения, зависящие от браузера, о том, каким должно быть возвращаемое значение. Хотя в настоящее время эти возвращаемые значения совместимы, мы в течение длительного периода объясняли сообществу, что произошло и почему.

16.6. Дистанционный драйвер и, в частности, драйвер Firefox

Дистанционный WebDriver изначально представлял собой хорошо известный механизм RPC. С тех пор он превратился в один из ключевых механизмов, которые мы используем с целью уменьшить расходы на поддержку WebDriver за счет предоставления единообразного интерфейса для привязки к коду языковых сборок. Даже, несмотря на то, что мы перенесли в драйвер столько логики, сколько смогли изъять из привязок к языкам, если бы нам потребовалось чтобы каждый драйвер взаимодействовал через уникальный протокол, у нас до сих пор было бы много кода, который повторялся бы во всех языковых привязках.

Дистанционный протокол WebDriver используется везде, где нам нужно было взаимодействовать с экземпляром браузера, работающим в другом процессе. При создании этого протокола нужно было учесть ряд проблем. Большинство из них были техническими, но, поскольку это открытый код, также рассматривался и социальный аспект.

Любой механизм RPC состоит из двух частей: транспортного механизма и механизма кодирования. Мы знали, что, несмотря на то, что мы реализовывали дистанционный протокол WebDriver, мы для тех языков, которые мы хотели использовать в качестве клиентских, должны были поддерживать обе части. Первая итерация проекта была разработана как часть драйвера Firefox.

Mozilla и, следовательно, Firefox всегда рассматривается их разработчиками как мультиплатформенное приложение. Чтобы помочь разработчикам, Mozilla по примеру COM от Microsoft создала фреймворк XPCOM (кросс-платформенный COM), который позволяет собирать компоненты и соединять их воедино. Интерфейс XPCOM описывается с помощью языка IDL и есть привязки для языков C и JavaScript и для других языков. Поскольку XPCOM используется для создания Firefox, и т. к. XPCOM является привязкой к языку Javascript, в расширениях Firefox можно использовать объекты XPCOM.

В обычной Win32 COM допускаются интерфейсы, к которым можно обращаться дистанционно. Были планы добавить к XPCOM такую же самую возможность и, чтобы способствовать этому, Дарин Фишер (Darin Fisher) добавил реализацию XPCOM ServerSocket. Хотя планы для D-XPCOM никогда не были осуществлены, все еще есть этаrudиментарная инфраструктура, похожая на апдейкс. Мы воспользовались этим для того, чтобы внутри пользовательского расширения Firefox, содержащего всю логику управления браузером Firefox, создать очень простой сервер. Первоначально используемый протокол был текстовый и строчно-ориентирован, причем для кодирования строк использовалась кодировка UTF-2. Каждый запрос или ответ начался с номера, указывающего, сколько должно быть получено символов новой строки прежде, чем можно было принять решение о том, что запрос или ответ был полностью отправлен. Важно отметить, что эта схема была проста для реализации на языке Javascript, поскольку в SeaMonkey (на тот момент движок Javascript для Firefox) для внутреннего хранения строк Javascript использовались 16-битные целые числа без знака.

Хотя занятия с пользовательскими протоколами кодирования поверх низкоуровневых сокетов представляют собой интересный способ скоротать время, у них есть ряд недостатков. Для пользовательского протокола нет широко распространенных библиотек, так что для каждого языка, который мы хотели поддерживать, нужно было все реализовывать с самого нуля. Эта необходимость реализовывать большое количество кода менее приятна, чем щедрость разработчиков открытого кода, которые могут поучаствовать в разработке новых языковых привязок. Кроме того, хотя когда мы рассыпали только текстовые данные, строчно-ориентированный протокол был замечателен, с ним возникли проблемы, когда нам потребовалось рассыпать изображения (например, скриншоты).

Очень быстро стало совершенно очевидно, что на практике этот исходный механизм RPC совершенно не годится. К счастью, был хорошо известен транспортный механизм, который широко

распространен, поддерживался почти на всех языках и который мог бы позволить нам делать все, что мы хотели: HTTP.

После того, как только мы решили в качестве транспортного механизма использовать HTTP, следующий выбор, который нужно было сделать, это решить, использовать ли механизм одного единого обращения (а-ля SOAP) или нескольких обращений (в стиле REST). В исходном протоколе Selenese использовался механизм единого обращения, в котором в строке запроса были закодированы команды и аргументы. Хотя этот подход работал хорошо, не «чувствовалось», что он правильный: мы предвидели, что нам для просмотра состояния сервера потребуется в браузере дистанционно подключаться к экземпляру WebDriver. Мы ограничились тем, что выбрали подход, который мы назвали «похожим на REST» («REST-ish»): множественные адреса URL, использующие операции HTTP, которые делают их его осмыслившими, но нарушают ряд ограничений, требующихся для истинных систем вида RESTful, в частности связанные с определением состояния и кэширования, главным образом постольку, поскольку есть только одно определение состояния приложения с тем, что в нем был смысл.

Хотя HTTP позволяет легко поддерживать множество вариантов кодирования данных, использующих соглашение о типе содержимого, мы решили, что нам нужен канонический вариант, с которым могли бы работать все реализации дистанционного протокола WebDriver. Было несколько очевидных вариантов: HTML, XML или JSON. Мы быстро исключили XML: хотя это вполне разумный формат данных и почти в каждом языке есть библиотеки, которые его поддерживают, у меня сложилось впечатление, что сообществу, работающему с открытым кодом, с ним работать не нравится. Кроме того, что хотя возвращаемые данные должны иметь обобщенный «вид», исключительно просто добавлять дополнительные поля [3]. Хотя такие расширения можно смоделировать с помощью пространства имен XML, с этого начинается постепенное усложнение клиентского кода: то, чего я стремился избежать. Вариант использования XML был отвергнут. HTML был не очень хорошим вариантом, поскольку нам требовалось определять наши собственные форматы данных, и хотя для этого можно было использовать встроенные механизмы микроформатов, это выглядело как использование молотка для разбивания яиц.

В качестве окончательной возможности был рассмотрен JavaScript Object Notation (JSON). Браузеры могут преобразовывать строку в объект либо при помощи прямого обращения к функции eval, либо на более современных браузерах с помощью примитивов, созданных для безопасного преобразования объектов языка Javascript в строку и наоборот без побочных эффектов. С практической точки зрения, JSON является популярным форматом данных, причем библиотеки для его обработки доступны почти для каждого языка и он нравится всем начинающим разработчикам. Легкий выбор.

Поэтому во второй итерации дистанционного протокола WebDriver в качестве транспортного механизма был использован HTTP, а в качестве схемы кодирования, используемой по умолчанию, был выбран JSON в кодировке UTF-8. UTF-8 был выбрана в качестве кодировки, используемой по умолчанию с тем, чтобы клиенты могли легко писать на языках с ограниченной поддержкой Unicode, т. к. в UTF-8 есть обратная совместимость с ASCII. Команды отправлялись на сервер по URL, в котором определялось, какая команда посыпается, а закодированные параметры команды указывались в массиве.

Например, вызов `WebDriver.get("http://www.example.com")` отображался в запрос POST в адресе URL, в котором закодирован идентификатор сессии, заканчивающийся на "/url", а массив параметров выглядит, например, как `{[]'http://www.example.com'[]}`. Возвращаемый результат имел несколько более сложную структуру и содержал информацию о месте, куда происходит возврат, и код ошибки. Это длилось недолго - до третьей итерации дистанционного протокола, в которой массив параметров, используемый в запросе, был заменен словарем именованных параметров. Преимущество такого подхода было в том, что запросы, используемые при отладке, были существенно проще, и клиент теперь не мог по ошибке неправильно указать порядок следования параметров, что сделало систему в целом более надежной. Естественно, было решено в тех случаях,

где это было делать наиболее уместно, использовать обычные коды ошибок HTTP для индикации определенных возвращаемых значений и ответов; например, если пользователь пытается вызвать URL, к которому нечего не отображено, или когда мы хотим указать, что «ответ не содержит данные».

Дистанционный протокол WebDriver имеет два уровня обработки ошибок: один - для неправильных запросов и один - для неверных команд. Например, неправильный запрос ресурса, которого нет на сервере, или, возможно, запрос действия, которое непонятно ресурсу (например, отправка команды DELETE — УДАЛИТЬ ресурсу, используемому для работы с адресом URL текущей страницы). В подобных случаях отсылается обычный ответ HTTP 4xx. Для команд, которые не удалось выполнить, возвращаемый код ошибки устанавливается равным 500 («Internal Server Error» — «Внутренняя ошибка сервера»), а возвращаемые данные содержат более подробные сведения о том, что пошло не так.

Когда с сервера посылаются данные, содержащие ответ, они имеют вид объекта JSON:

Ключ	Описание
sessionId	Скрытый регулятор, используемый сервером для того, чтобы определить, куда направлять команды конкретной сессии.
status	Цифровой код состояния, подытоживающий результат работы команды. Ненулевое значение указывает, что команда не была выполнена.
value	Значение - ответ JSON.

Например, ответ может быть следующим:

```
{
  sessionId: 'BD204170-1A52-49C2-A6F8-872D127E7AE8',
  status: 7,
  value: 'Unable to locate element with id: foo'
}
```

Видно, что в ответе закодирован код состояния с ненулевым значением, указывающим, что произошло что-то ужасно непоправимое. Драйвер IE был первым, где использовались коды состояния, и это отразилось на значениях, используемых в сетевом протоколе. Поскольку все коды ошибок во всех драйверах согласованы между собой, можно код, обрабатывающий ошибки, использовать во всех драйверах, написанных на определенном языке, что упрощает работу разработчиков на клиентской стороне.

Сервер дистанционной обработки Remote WebDriver Server является простым сервлетом на языке Java, который работает как мультиплексор, выполняющий перенаправление всех команд, которые он получает, соответствующему экземпляру WebDriver. Это то, что мог бы написать аспирант второго года обучения. В драйвере Firefox также реализован протокол дистанционного доступа WebDriver, и существенно более интересна архитектура этого драйвера, так что давайте проследим за запросом, который поступает из вызова в языковой привязке в работающий в фоновом режиме драйвер, который затем возвращает пользователю ответ.

Предположим, что мы используем язык Java и что элементом «element» является экземпляр WebElement, тогда все начинается следующим образом:

```
element.getAttribute("row");
```

Если заглянуть внутрь, то в элементе есть идентификатор «id», который на серверной стороне используется для идентификации элемента, о котором идет речь. В рамках этого обсуждения мы будем считать, что он имеет значение «некоторое_скрытое_id». Оно кодируется в Java-объекте

Command с помощью класса Map, хранящего (на этот раз именованные) параметры id - для элемента ID и name - для имени запрашиваемого атрибута.

Беглый взгляд в таблицу указывает, что правильным адресом URL будет следующий:

```
/session/:sessionId/element/:id/attribute/:name
```

Любая часть адреса URL, которая начинается с двоеточия, считается переменной, для которой необходимо подставить значение. У нас уже заданы параметры id и name, а идентификатор сессии sessionId является еще одним скрытым регулятором, который используется при маршрутизации в случае, если сервер одновременно обрабатывает более одной сессии (что не может делать драйвер Firefox). Поэтому такой адрес URL обычно раскрывается, например, следующим образом:

```
http://localhost:7055/hub/session/XXX/element/some_opaque_id/attribute/row
```

В качестве отступления: протокол дистанционного доступа для WebDriver был первоначально разработан в то же самое время, когда в качестве чернового варианта RFC были предложены шаблоны URL Templates. Обе наши схемы - спецификация адресов URL и шаблоны URL Templates - позволяют в адресе URL использовать переменные (и, следовательно, вычислять значения). Хотя шаблоны URL Templates предлагались в то же время, нам, к сожалению, стало о них известно довольно поздно, и поэтому при описании сетевого протокола они не использовались.

Поскольку метод мы, который мы выполняем, идемпотентен [4], правильным методом HTTP, который следует использовать, будет метод GET. Мы все делегировали в библиотеку Java, которая может обрабатывать запросы HTTP (клиент Apache HTTP), направляемые в сервер.

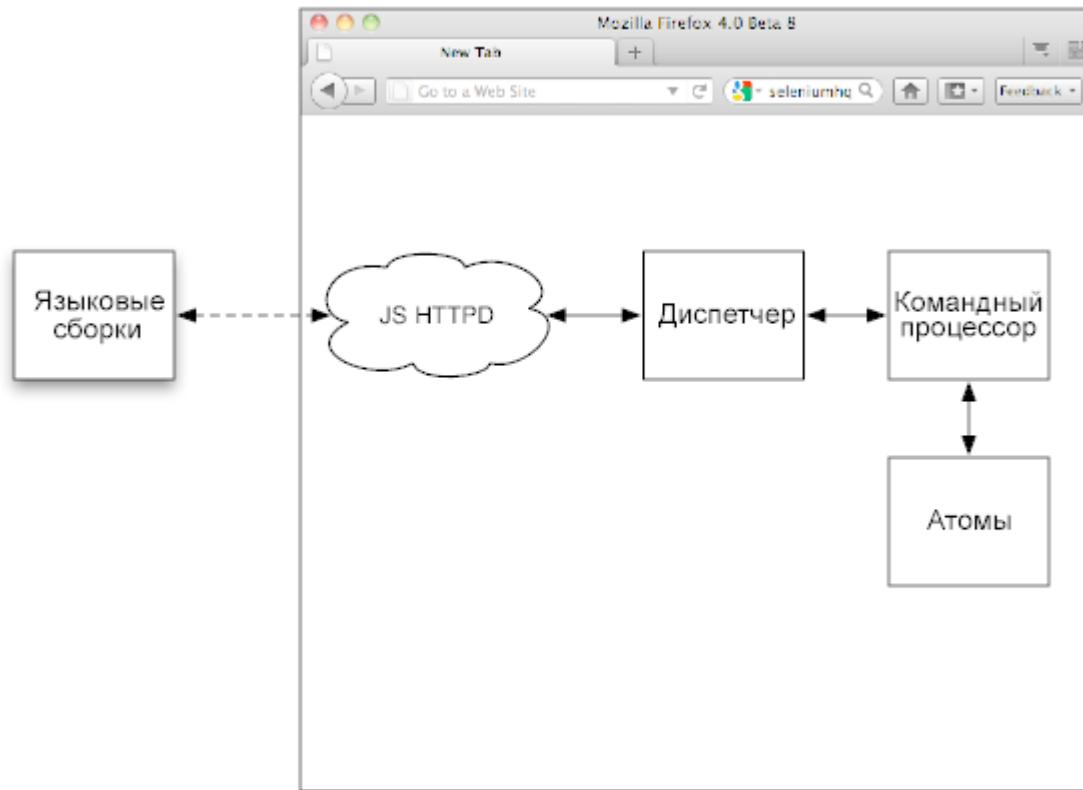


Рис.16.4: Общая схема архитектуры драйвера Firefox

Драйвер Firefox реализован как расширение браузера Firefox; базовая архитектура драйвера показана на рис.16.4. Несколько необычно то, что имеется встроенный сервер HTTP. Хотя первоначально мы использовали тот сервер, который мы сделали самостоятельно, создание серверов HTTP в XPCOM не относилось к тем задачам, в которых мы компетентны, и когда появилась воз-

можность, мы заменили его простейшим сервером HTTPD, написанным в рамках самого проекта Mozilla. Запросы поступают на сервер HTTPD и почти сразу же передаются объекту-диспетчеру dispatcher.

Диспетчер принимает запрос и перебирает известный ему список поддерживаемых адресов URL, пытаясь найти тот, который соответствует запросу. Это сравнение выполняется с учетом вставки значения переменной, которое выполнено на клиентской стороне. Как только будет найдено точное совпадение, в том числе и выполнение действия, будет сконструирован объект JSON, представляющий собой команду для запуска. В нашем случае это выглядит следующим образом:

```
{
  'name': 'getElementAttribute',
  'sessionId': { 'value': 'XXX' },
  'parameters': {
    'id': 'some_opaque_key',
    'name': 'rows'
  }
}
```

Затем это будет передано в виде строки JSON в пользовательскую компоненту XPCOM, написанную нами, которая называется командным процессором CommandProcessor. Ее код следующий:

```
var jsonResponseString = JSON.stringify(json);
var callback = function(jsonResponseString) {
  var jsonResponse = JSON.parse(jsonResponseString);

  if (jsonResponse.status != ErrorCode.SUCCESS) {
    response.setStatus(Response.INTERNAL_ERROR);
  }

  response.setContentType('application/json');
  response.setBody(jsonResponseString);
  response.commit();
};

// Dispatch the command.
Components.classes['@googlecode.com/webdriver/command-processor;1'].
  getService(Components.interfaces.nsICommandProcessor).
  execute(jsonString, callback);
```

Здесь сравнительно много кода, но есть два ключевых момента. Во-первых, мы преобразовали объект, рассмотренный выше, в строку JSON. Во-вторых, мы передали в метод execute функцию обратного вызова, которая будет выполнена, когда будет отправлен ответ HTTP.

Метод execute котмандного процессора ищет «имя» чтобы определить, какую функцию нужно вызывать, что затем он и делает. Первый параметр передаваемый для этой функции, является объектом «respond» (так называется потому, что первоначально он просто использовался функцией для возврата ответа обратно к пользователю), в котором инкапсулированы не только те возможные значения, которые могут быть посланы, но также метод, который позволяет перенаправить ответ обратно пользователю, и механизмы, позволяющим узнать информацию о модели DOM. Вторым параметром является значение объекта parameters, который мы уже видели раньше (в данном случае, id и name). Преимущество этой схемы в том, что во всех функциях используется единообразный интерфейс, в котором отражена структура, используемая на клиентской стороне. Это значит, что на каждой стороне ментальные модели, используемые для рассмотрения кода, аналогичны. Ниже приведена реализация метода getAttribute, который вы уже видели в разделе 16.5:

```
FirefoxDriver.prototype.getAttribute = function(respond, parameters) {
  var element = Utils.getElementAt(parameters.id,
    respond.session.getDocument());
  var attributeName = parameters.name;
```

```

    respond.value = webdriver.element.getAttribute(element, attributeName);
    respond.send();
}

```

Чтобы сделать ссылки на элемент согласованными, в первой строке просто выполняется поиск элемента, на который указывает скрытый идентификатор ID, находящийся в кэше. В драйвере Firefox, этот скрытый ID является просто UUID, а «кэш» является просто отображением. Метод getElementAt также проверяет, известен ли этот элемент и есть ли он в модели DOM. Если какая-либо из проверок оказалась неудачной, то ID удаляется из кэша (если это необходимо) и пользователю возвращается исключительное состояние.

Во второй строчке от конца используются атомы автоматизации браузера, рассмотренные ранее, которые на этот раз откомпилированы в виде монолитного скрипта и загружены как часть расширения.

В последней строке, вызывается метод send. Здесь делается простая проверка с тем, чтобы перед тем, как вызвать функцию обратного вызова, указанную в методе execute, удостовериться, что мы отправляем ответ только один раз. Ответ отправляется обратно пользователю в виде строки JSON, преобразуемой в объект, который выглядит следующим образом (при условии, что getAttribute возвращает «7», означающий, что элемент не найден):

```
{
  'value': '7',
  'status': 0,
  'sessionId': 'xxx'
}
```

Затем клиент языка Java проверяет значение поля состояния. Если это значение не равно нулю, он, используя поле значения «value», которое помогает установить значение, посланное пользователю, преобразует цифровой код состояния в исключительное состояние соответствующего типа. Если состояние равно нулю, то значение поля «value» возвращается пользователю.

Большая часть того, что происходит, понятно, но есть одна вещь, о которой проницательный читатель может спросить: почему диспетчер перед тем, как вызвать метод execute, преобразует объект, который у него есть, в строку?

Причина этого в том, что драйвер Firefox также поддерживает выполнение тестов, написанных на чистом языке Javascript. Обычно осуществлять такую поддержку чрезвычайно трудно: тесты выполняются в браузере в контексте песочницы безопасности Javascript, и, поэтому, не удается выполнять целый ряд действий, которые могли бы использоваться в тестах, например, переход между доменами или загрузка файлов на сервер. Однако расширение Firefox для WebDriver предоставляет возможность выхода из песочницы. Оно объявляет о своем присутствии при помощи добавления к элементу document свойства webdriver. Интерфейс Javascript API для WebDriver использует это свойство как индикатор того, что он может добавить к элементу document сериализованные объекты команд JSON как значение свойства command, включать событие webdriverCommand, а затем для того же самого элемента ожидать события webdriverResponse, которое сообщает о том, что свойство response было установлено.

При этом предполагается, что просмотр интернета с помощью копии Firefox с установленным расширением WebDriver — действительно плохая мысль, поскольку кому-то другому станет исключительно просто дистанционно управлять браузером.

За всем этим есть мессенджер DOM, ожидающий, когда webdriverCommand прочитает сериализованный объект JSON и вызовет в командном процессоре метод execute. На этот раз, обратный вызов будет просто установкой атрибута response для элемента document, а затем отработкой ожидаемого события webdriverResponse.

16.7. Драйвер IE

Internet Explorer является интересным браузером. Он сконструирован из ряда интерфейсов COM, работающих как единый оркестр. Это происходит посреду в движке Javascript, где знакомые нам переменные языка Javascript на самом деле ссылаются к лежащим в их основе экземплярам COM. Переменная `window` в Javascript является IHTMLWindow. Переменная `document` является экземпляром COM-интерфейса IHTMLDocument. Фирма Microsoft, в процессе разработки своих браузеров, проделала отличную работу по сохранении существующего поведения интерфейсов COM. Это означает, что если приложение работает с COM-классами, предоставляемыми IE6, то оно будет также продолжать работать с IE9.

Драйвер Internet Explorer имеет архитектуру, которая развивалась на протяжении длительного времени. Одним из основных направлений затрат усилий на ее разработку было соблюдение требования не использовать инсталлятор. Это немного необычное требование, поэтому, возможно, требуется некоторое пояснение. Первая причина не использовать инсталлятор объясняется тем, что он осложняет драйверу WebDriver проходит «5 минутный тест», когда разработчик загружает пакет и испытывает его в течение непродолжительного периода времени. Более того, у пользователей WebDriver относительно типичная ситуация, когда на их машинах нельзя устанавливать программное обеспечение. Это также означает, что когда нужно начинать тестирование проекта с использованием IE, никто не должен помнить, как входить на постоянно работающий сервер с тем, чтобы запустить инсталлятор. Наконец, запускаемые инсталляторы просто не соответствуют культуре использования некоторых языков. Обычная идиома языка Java состоит просто в указании файлов JAR в переменной CLASSPATH, и, по моему опыту, те библиотеки, для которых требуются инсталляторы, как правило, не столь любими и не столь широко применимы.

Итак, инсталлятора нет. И это решение ведет к следующему следствию.

Типичным языком, используемым для программирования на Windows, может быть любой, которые работает на платформе .Net, например, C#. Драйвер IE интегрируется с IE с помощью интерфейсов автоматизации IE COM Automation, которые поставляются с каждой версией Windows. В частности, мы используем интерфейсы COM из нативных DLL-библиотек MSHTML и ShDocVw, которые являются частью IE. До C# 4, взаимодействие CLR/COM достигалось за счет использования отдельных сборок Primary Interop Assemblies (сборок PIA). Сборка PIA является, в сущности, сгенерированным мостиком между управляемым миром CLR и миром COM.

К несчастью, использование C# 4 будет означать использование очень современной среды исполнения .Net, а многие компании избегают жить на переднем крае разработок, предпочитая стабильность и известные проблемы более старых версий. При использовании C# 4 мы автоматически исключаем определенный процент нашей базы пользователей. Есть и другие недостатки в использовании PIA. Рассмотрим лицензионные ограничения. После консультаций с Microsoft, стало ясно, что проект Selenium не будет иметь прав на распространение сборок PIA ни в виде библиотек MSHTML, ни в виде библиотек ShDocVw. Даже если эти права были бы предоставлены, каждая установка Windows и IE имеет уникальное сочетание этих библиотек, а это означает, что нужно было бы поставлять огромное количество лишних библиотек. Создание сборок PIA по требованию на клиентской машине также не для начинающих разработчиков, поскольку для этого требуются инструментальные средства, которых может не быть на компьютере обычного пользователя.

Поэтому, хотя могло бы быть заманчиво использовать язык C# для создания большей части кода, он не подходил. Нам нужно было использовать что-то естественное, по крайней мере, для взаимодействия с IE. Следующим естественным выбором для этого стал язык C++, и это был тот язык, который мы, в конце концов, выбрали. При использовании языка C++ у нас преимущество в том, что не нужно использовать сборки PIA, но это означает, что в случае, если мы не выполняем с библиотекой Visual Studio C++ Runtime DLL статическую компоновку, мы должны ее распро-

странять. Поскольку для того, чтобы библиотека DLL была доступна, нам требуется запускать инсталлятор, мы статически компонуем нашу библиотеку для взаимодействия с IE.

Т.е. цену, которую нужно заплатить за требование не использовать инсталлятор, слишком высокая. Но, если вернуться к теме, где явно видна вся сложность, то в ней инвестиции стоящие, поскольку они делают жизнь наших пользователей значительно проще. Это решение, которое мы пересмотрели, поскольку преимущества для пользователя является компромиссом с тем, что, как оказывается, количество тех, кто способен внести свой вклад в современный проект на языке C++ с открытым исходным кодом значительно меньше, чем тех, кто может внести свой вклад в эквивалентный проект на языке C#.

На рис.16.5 показана исходная схема драйвера IE

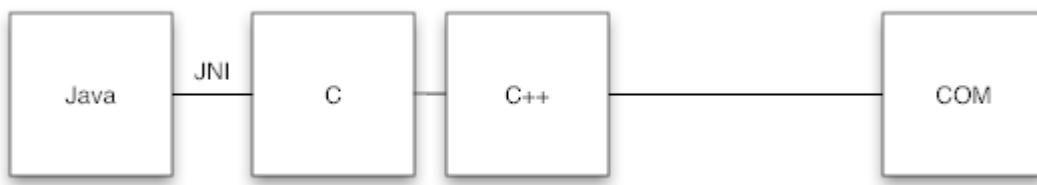


Рис.16.5: Исходная схема драйвера IE

Если начать с самой правой части этой цепочки, то видно, что мы используем интерфейсы автоматизации IE - COM Automation. Для того, чтобы с ними на концептуальном уровне было легче иметь дело, мы поместили эти интерфейсы внутрь набора классов C++, которые непосредственно отображаются в WebDriver API. Чтобы позволить классам языка Java взаимодействовать с C++, мы воспользовались технологией JNI, причем при реализации методов JNI использовались абстракции C++ интерфейсов COM.

Пока язык Java был единственным языковым клиентом, этот подход работал достаточно хорошо, но в случае, если для каждого языка, который нам требовалось поддерживать, нужно было заменять базовую библиотеку, он мог бы быть источником боли и сложности. Итак, хотя технология JNI работала, она не обеспечивала необходимый уровень абстракции.

Что было правильным уровнем абстракции? В каждом языке, который нам требовалось поддерживать, был механизм непосредственного вызова кода на языке С. В языке C#, он имеет вид PInvoke. В языке Ruby есть FFI, а в языке Python есть ctypes. В мире Java, есть отличная библиотека, которая называется JNA (Java Native Architecture — нативная архитектура языка Java). Нам нужно было предоставить наш интерфейс API с помощью этого наименьшего общего деноминатора. Это было сделано путем преобразования нашей объектной модели в более плоскую модель, использующую двух или трехбуквенный префикс, которые указывают на «домашний интерфейс» метода: wd - для WebDriver и wde - для элемента WebDriver Element. Таким образом WebDriver.get стал wdGet, а WebElement.getText стал wdeGetText. Каждый метод возвращает целое число, представляющее собой код состояния, и параметры out, позволяющие функциям возвращать более конкретные данные. Таким образом, мы пришли к сигнатурами методов, например:

```
int wdeGetAttribute(WebDriver*, WebElement*, const wchar_t*, StringWrapper**)
```

При вызове кода, типы сигнатур WebDriver, WebElement и StringWrapper не показываются: для того, чтобы было понятно, какое значение должно использоваться в качестве конкретного параметра, мы отразили различие в интерфейсе API, хотя проще было бы использовать описатель «void *». Также вам должно быть видно, что для текста мы использовали символы расширенного формата, т. к. нам нужно было работать с текстовыми свойствами на различных языках.

На стороне языка Java мы предоставили доступ к функциям этой библиотеки через интерфейс, который затем мы адаптировали так, что он стал выглядеть как обычный объектно-ориентированный интерфейс, представляемый пакетом WebDriver. Например, определение на языке Java метода `getAttribute` выглядит следующим образом:

```
public String getAttribute(String name) {
    PointerByReference wrapper = new PointerByReference();
    int result = lib.wdeGetAttribute(
        parent.getDriverPointer(), element, new WString(name), wrapper);

    errors.verifyErrorCode(result, "get attribute of");

    return wrapper.getValue() == null ? null : new StringWrapper(lib, wrapper).toString();
}
```

Это привело к использованию схемы, показанной на рис.16.6.

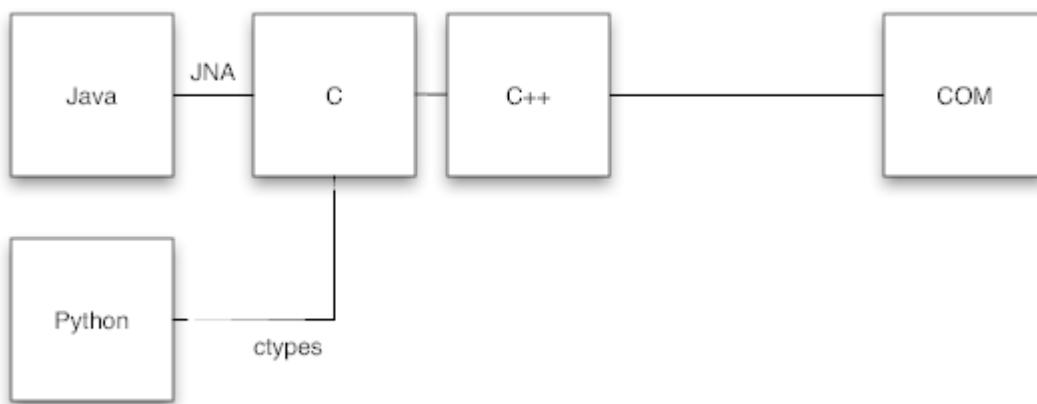


Рис.16.6: Измененная схема драйвера IE

Пока все тесты исполнялись на локальном компьютере, все это работало хорошо, но как только мы начали использовать драйвер IE в для WebDriver дистанционно, у нас стали в случайном порядке возникать блокировки. Мы проследили эту проблему и обнаружили ограничение в интерфейсах автоматизации IE COM Automation. Эти интерфейсы предназначены для использования в «однопотоковой» модели. В сущности все это сводится к требованию, чтобы интерфейс каждый раз вызывался из того же самого потока. Когда запуск осуществляется на локальной машине, то выполняется по умолчанию. Однако сервера приложений Java для того, что распределить возможную нагрузку, и запускают несколько потоков. Что в итоге? У нас не было способа, чтобы всегда быть уверенным, что для доступа к драйверу IE будет использоваться один и тот же поток.

Одним из способов решения этой проблемы мог бы быть запуск драйвера IE с помощью модуля, осуществляющего его запуск в одном и том же потоке, и сериализация всего доступа через Futures сервера приложений, и на некоторое время мы выбрали эту схему. Но, как оказалось, нагружать всей этой сложностьюзывающий код является неправильным решением, причем все равно слишком легко представить себе ситуации, когда драйвер IE будут случайно использовать из нескольких потоков. Мы решили упрятать эту сложность поглубже в самом драйвере. Мы сделали это, поместив экземпляр IE в отдельный поток и использовав PostThreadMessage из интерфейса Win32 API для передачи сообщений через границу потока. Таким образом, на момент написания статьи, схема драйвера IE выглядит так, как это показано на рис.16.7.

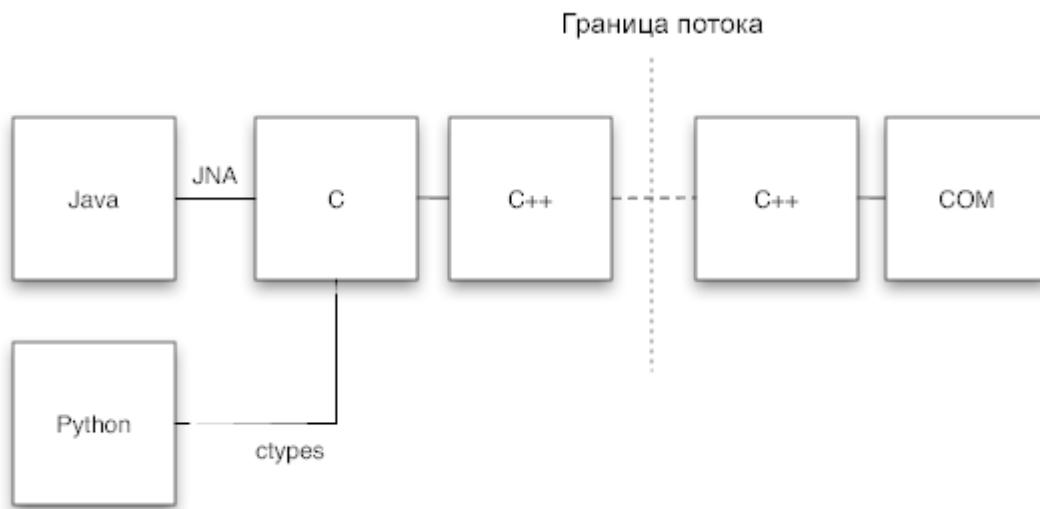


Рис.16.7: Схема драйвера IE версии Selenium 2.0 alpha 7

Это не та схема, которую я бы выбрал добровольно, но она дает возможность работать и выдерживать те ужасы, которые наши пользователи могут в нее добавить.

Один из недостатков этой схемы заключается в том, что трудно определить, намерто ли заблокировал сам себя экземпляр IE. Это может произойти в случае, если во время взаимодействия с DOM открывается модальное диалоговое окно, либо это может произойти в случае, если катастрофическая ситуация возникнет с другой стороны границы потока. Поэтому мы имеем таймаут, связанный с каждый сообщением в потоке, которое мы передаем, и он устанавливается равным относительно щедрым 2 минутам. По обратной реакции пользователей, получаемой из списка рассылки, это допущение, хотя в целом и верно, но не всегда правильное, и в более поздних версиях драйвера IE, вполне возможно, таймаут можно будет задавать самостоятельно.

Еще один недостаток состоит в том, что отладка внутренних механизмов кода может стать крайне проблематичной, требующей сочетания скорости (в конце концов, у вас есть две минуты, чтобы оттрассировать настолько, насколько это будет возможным), разумного использования точек останова и понимание предполагаемого пути, где будет происходить пересечение границы потока. Излишне говорить, что в проекте с открытым кодом и со многими другими интересными проблемами, которые необходимо решать, с шероховатостями такого рода возиться совсем неинтересно. В системе это существенно сокращает фактор автобуса и меня, как менеджера проекта, это беспокоит.

Чтобы с этим справиться, драйвер IE точно также как и драйвер Firefox и Selenium Core, все больше и больше перемещается туда, где находятся атомы автоматизации Automation Atoms. Мы делаем это с помощью компилирования каждого из атомов, который мы планируем использовать, и подготовки его в виде заголовочного файла C++, где каждая функция представлена в виде константы. Мы подготавливаем Javascript с тем, чтобы во время выполнения он мог выполнять эти константы. Такой подход означает, что мы можем разработать и протестировать значительное количество кода в драйвере IE без необходимости привлечения компилятора C, а это, в свою очередь, позволяет гораздо большему числу разработчиков внести свой собственный вклад в поиск и исправление ошибок. Целью, в конце концов, является оставить в нативном виде только интерфейсы API, используемые при взаимодействии, и как можно больше работы возложить на атомы.

Еще один подход, которые мы исследуем, состоящий в переписывании драйвера IE для того, чтобы добавить легкий сервер HTTP, позволяющий нам использовать его как WebDriver, работающий дистанционно. Если это произойдет, то мы сможем избавиться от многих сложностей, связанных с границами потоков, сократить общее количество необходимого кода и сделать процесс управления значительно проще.

16.8. Selenium RC

Не всегда можно подключиться непосредственно к конкретному браузеру. В таких случаях, WebDriver возвращается к исходному механизму, используемому в Selenium. Это означает использование чистого Javascript-фреймворка Selenium Core, у которого есть ряд недостатков, т. к. он выполняется исключительно в контексте песочницы Javascript. В отличие от использования интерфейса WebDriver API это означает, что сразу сильно сокращается список поддерживаемых браузеров, причем интеграция с некоторыми из них очень хорошая и имеются исключительные возможности управления, а для других, управляемыми через Javascript, реализован точно такой же уровень управления, что и в исходном Selenium RC.

На рис.16.8 показывается, что концептуально используемая схема является сравнительно простой.



Рис.16.8: Общая схема архитектуры Selenium RC

Видно, что здесь есть три части: код клиента, промежуточный сервер и код Javascript-фреймворка Selenium Core, работающий в браузере. Со стороны клиента это только HTTP-клиент, который сериализует команды, посылаемые на серверную сторону. В отличие от дистанционно работающего WebDriver, здесь есть только одноточечный запрос, а выполняемое действие HTTP большой роли не играет. Это отчасти потому, что протокол Selenium RC создан на основе табличного интерфейса API, предоставляемого в Selenium Core, а это значит, что весь интерфейс API можно описать в адресе URL с помощью трех параметров запроса.

Когда клиент запускает новую сессию, сервер Selenium ищет в запросе «строку браузера» для того, чтобы выбрать лаунчер (программу запуска — прим.пер.), соответствующий браузеру. На лаунчере возложена обязанность сконфигурировать и запустить экземпляр запрашиваемого браузера. В случае с Firefox, это столь же просто, как раскрыть встроенный профиль с несколькими предварительно установленными расширениями (одно — для обработки команды выхода quit, а другое — для моделирования состояния готовности document.readyState, отсутствующее в старых релизах Firefox, которые мы до сих пор поддерживаем). Ключевая часть выполняемого конфигурирования представляет собой настройку сервера для его использования в качестве прокси для браузера, что означает, что через него проходят по крайней мере некоторые запросы (те, что для /selenium-server). Selenium RC может работать в одном из трех режимов: управление фреймом в единственном окне (режим одного окна singlewindow), управления AUT в отдельном втором окне (много-

оконный режим multiwindow) или внедрение в страницы через прокси-сервер (режим прокси-инъекции proxyinjection). В зависимости от того, какой режим работы выбран, через прокси могут проходить все запросы.

Когда браузер сконфигурирован, он запускается с первоначальным адресом URL, указывающим на страницу, размещенную на сервере Selenium - RemoteRunner.html. Эта страница отвечает за развертывание процесса загрузки всех файлов Javascript, необходимых для Selenium Core. После завершения загрузки вызывается функция runSeleniumTest, запускающая тестирование. Здесь объект Selenium используется для инициализации списка команд, которыми можно воспользоваться перед запуском основного цикла обработки команд.

Javascript, исполняемый в браузере, отправляет по URL запрос XMLHttpRequest ожидающему серверу (/selenium-server/driver), при этом предполагается, что сервер является прокси-сервером для всех запросов, тем самым позволяя проверять, что запрос, куда бы он не был направлен, является допустимым. Вместо того, чтобы делать запрос, первое, что делается, это отправляется ответ на ранее выполненную команду или OK в случае, если браузер был только что запущен. Затем сервер поддерживает запрос открытый до тех пор, пока от теста, выполняемого пользователем, не будет через клиентскую часть получена новая команда, которая затем посыпается в качестве ответа для ожидающего Javascript. Этот механизм первоначально был назван «Response/Request» («ответ/запрос»), но теперь его было бы правильнее назвать «Comet with AJAX long polling» (Comet с пролонгированным AJAX-запросом).

Почему RC работает таким образом? Сервер нужно настроить как прокси-сервер для того, чтобы он мог перехватывать любые запросы, которые делаются, и не обращался к JavaScript, т. к. последнее нарушает политику «Single Host Origin», которая гласит, что с помощью Javascript можно запрашивать ресурсы только с того же сервера, откуда был получен скрипт. Это одна из мер безопасности, но, с точки зрения разработчика фреймворка автоматизации браузера, эта мера является проблемой и ее необходимо обходить.

Запрос XMLHttpRequest отправляется на сервер по следующим двум причинам. Первая, и самая главная, что пока сокеты WebSockets , как часть HTML5, не стали доступными в большинстве браузеров, нет способа из браузера гарантированно запустить процесс на сервере. Это значит, что сервер нужно оживлять как-нибудь иначе. Вторая причина в том, запрос XMLHttpRequest запускает функцию обратного вызова ответа асинхронно, а это означает, что пока мы ждем следующую команду, ничего другое не изменит обычное исполнение браузера. Двумя другими способами ожидания следующей команды мог бы быть регулярный опрос сервера с тем, чтобы посмотреть, нет ли на выполнение другой команды, из-за чего в пользовательских тестах возникали бы задержки, или можно было бы поместить команду Javascript в цикл с большой нагрузкой для того, чтобы он через край загрузил процессор и помешал бы в браузере выполнению другой команды Javascript (поскольку в контексте одного окна выполняется только один поток Javascript).

Внутри Selenium Core есть две основные части. Это главный объект selenium, который для всех имеющихся команд выступает в роли хоста и отображает интерфейс API, предлагаемый пользователям. Второй частью является бот browserbot. Он используется объектом selenium для абстрагирования от различий, имеющихся в каждом браузере, и представляет собой идеализированное представление об обычных функциональных возможностях браузера. Это означает, что функции в объекте selenium становятся более понятными и их легче поддерживать, а все особенности сосредоточены в browserbot.

Для того, чтобы можно было использовать атомы автоматизации Automation Atoms, компонента Core все изменяется все больше и больше. Обе части selenium и browserbot, вероятно, должны остаться, поскольку есть достаточно много кода, использующего интерфейс API, который они предоставляют, но предполагается, что, в конечном итоге, они станут оболочками классов, осуществляющих перенаправление к атомам настолько быстро, насколько это будет возможным.

16.9. Оглядываясь назад

Создание фреймворка автоматизации браузера во многом похоже на рисование комнаты; на первый взгляд, это выглядит как нечто, что должно делаться довольно легко. Все, что требуется сделать - это несколько слоев краски и работа выполнена. Проблема в том, что чем ближе вы подходитите, тем больше появляются задач и деталей и тем дольше будет выполнение задачи. Что касается комнаты, это что-то вроде рисования ламп и светильников, радиаторов и плинтусов, на что начинает тратиться время. Что касается автоматизации браузеров, то это те все особенности и различия в возможностях браузеров, которые делают ситуацию еще более сложной. Самим резким образом на это отреагировал Даниэль Вагнер-Холл (Daniel Wagner-Hall), когда он, сидя рядом со мной и работая над драйвером Chrome, ударил руками по столу и в отчаянии пробормотал: «Все это - крайние случаи!». Было бы хорошо, если бы можно было вернуться в прошлое и сообщить самому себе о том, что проект займет намного больше времени, чем я ожидал.

Я также не могу не спросить, чтобы бы было с проектом, если бы мы определялись и делали только то, что необходимо в слое, например, атомы автоматизации ранее, чем мы это сделали. Конечно, были бы решены некоторые из проблем, с которыми столкнулся проект, касающихся внутренних и внешних, технических и социальных аспектов. Core и RC были реализованы для ограниченного набора языков, по сути только для Javascript и Java. Джейсон Хаггинс Huggins (Jason Huggins) говорил об этом как об уровне «осваиваемости», присутствующим в Selenium, который делает проект легким для тех, кто принимает в нем участие. Этот уровень «осваиваемости» стал везде доступным в WebDriver только благодаря атомам. А что касается последнего — причина, по которой стало возможным так широко использовать атомы в том, что компилятор Closure Compiler, который мы стали использовать почти сразу, был выпущен как проект с открытым кодом.

Также интересно поразмышлять о том, что мы сделали правильно. Я все еще считаю, что написать то, что с точки зрения пользователя, выглядит как фреймворк, является правильным решение. Сначала это окупилось тем, что сразу были найдены места, где потребовались улучшения, и это быстро увеличило отдачу от инструментального средства. Позже, когда драйвер WebDriver сталправляться с более крупными и сложными задачами, а число разработчиков, его использующих, увеличилось, это значило, что новые интерфейсы API нужно было добавлять аккуратно и внимательно, строго сохраняя целенаправленность проекта. Учитывая масштабы того, что мы пытаемся сделать, эта целенаправленность является жизненно важной.

Жесткая привязка к браузеру является в некотором смысле одновременно как правильной, так и неправильной. Она правильная, поскольку позволила нам с особой точностью имитировать поведение пользователей и очень хорошо управлять браузером. Она неправильная, поскольку такой подход является технически чрезвычайно сложным, особенно когда в браузере для подключения нужно найти нужный механизм. Действующим примером является постоянная эволюция драйвера IE и, хотя этого мы здесь не рассматривали, то же самое можно сказать и о драйвере Chrome, который имеет свою долгую и интересную историю. В какой-то момент нам потребовалось найти способ справиться с этой сложностью.

16.10. Заглядывая в будущее

Всегда будут браузеры, с которыми драйверу WebDriver не удастся тесно интегрироваться, так что всегда будет необходимость в Selenium Core. Полным ходом идет переход из текущей традиционной формы в более модульную конструкции, в основе которой лежит та же самая библиотека Closure Library, которая используется с атомами. Также предполагается, что мы будем более глубоко внедрять использование атомов в уже существующих реализациях WebDriver.

Одной из первоначальных задач было использование WebDriver в качестве строительного модуля для других интерфейсов API и инструментальных средств. Конечно, Selenium разрабатывается не в вакууме: есть много других средств для автоматизации браузеров, имеющий открытый исход-

ный код. Одним из них является Watir (Web Application Testing In Ruby — тестирование веб приложений в языке Ruby), и была начата работа, представляющая собой совместные усилия разработчиков Selenium и Watir по использованию интерфейса Watir API поверх WebDriver. Мы также заинтересованы в работе с другими проектами, поскольку успешное управление всеми браузерами — это тяжелый труд. Было бы неплохо иметь твердую основу, на которую могли бы опираться другие. Мы надеемся, что этой основой является WebDriver.

Представление о будущем проекта было показано компанией Opera Software, которая самостоятельно реализовала интерфес WebDriver API, использующий набор тестов WebDriver для проверки работы своего собственного кода, и которая будет выпускать свой собственный драйвер OperaDriver. Разработчики Selenium также работают с разработчиками Chromium над вопросами добавления лучших механизмов подключения к этому браузеру и вопросами поддержки WebDriver для этого браузера, в том числе и с помощью расширений Chrome. У нас дружеские отношения с компанией Mozilla, которая предоставила свой код для драйвера FirefoxDriver, и с разработчиками популярного эмулятора браузеров HtmlUnit, используемого с языком Java.

Взгляд в будущее показывает, что эта тенденция продолжится — механизмы подключения, используемые для автоматизации, преобразуются во многих различных браузерах к единообразному виду. Ясно, что это преимущество для тех, кто занимается написанием тестов для веб-приложений, и также очевидно, что это преимущество для разработчиков браузеров. Например, из-за сравнительно дорого тестирования вручную, во многих крупных проектах в значительной степени полагаются на автоматизированное тестирование. Если оно невозможно, или даже если оно используется «всего лишь» тестирования с конкретным браузером, то тесты, не прошедшие с ним, показывают общую картину, насколько хорошо с этим браузером работают сложные приложения. Вопрос о том, будут ли эти механизмы автоматизации основываться на использовании WebDriver, остается открытым, но мы можем надеяться!

Ближайшие несколько лет будут очень интересными. Поскольку мы — проект с открытым кодом, мы приглашаем вас присоединиться к нам в нашем турне на странице <http://selenium.googlecode.com/>.

Примечания

1. <http://fit.c2.com>
2. Это очень похоже на проект FIT, и Джеймс Шор (James Shore), один из его координаторов, помог на странице <http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html> разобраться с некоторыми его недостатками
3. Например, сервер, работающий дистанционно, каждый раз, когда возникает исключение, возвращает копию экрана в кодировке base64 в качестве дополнительного средства отладки, а драйвер Firefox этого не делает
4. Т.е. всегда возвращает один и тот же результат.

17. Sendmail - Архитектура и принципы разработки

Под электронной почтой большинство людей подразумевает ту программу, которой они пользуются, то есть свой e-mail клиент. На техническом языке он называется «почтовым агентом пользователя». Но есть и другая важная часть электронной почты — программное обеспечение, которое в действительности осуществляет пересылку сообщений от отправителя к получателю — агент передачи почты (Mail Transfer Agent — MTA). Первой, и до сих пор наиболее используемой из подобных программ, стала программа sendmail.

Sendmail был создан еще до того, как Интернет официально начал свое существование. Программа стала феноменально успешной. Ее история ведет отсчет с 1981 года, когда еще совсем не было ясно, что Интернет станет чем-то большим, чем академический эксперимент с сотней хостов, и продолжается до сегодняшнего дня, когда число пользователей Сети перевалило за 800 миллионов (по состоянию на январь 2011). Sendmail остается одной из наиболее используемых реализаций протокола SMTP в Интернете.

17.1. Как все начиналось...

Первые версии программы, которая будет позже известна как sendmail, были написаны в 1980 году. Программа началась как быстрый хак для передачи сообщений между различными сетями (Интернет в то время разрабатывался, но еще не был полнофункциональным). Предполагалось наличие множества различных сетей, и не было единого консенсуса на тему, какая из сетей будет главной. В США использовалась Арпанет, а Интернет конструировался как ее апгрейд, однако Европа решительно настаивала на OSI (Open Systems Interconnect), и какое-то время казалось, что OSI может взять верх. Обе сети использовали выделенные линии телефонных компаний; в США скорость таких линий составляла 56 kbps.

Вероятно, наиболее успешной сетью на то время, с точки зрения числа подключенных компьютеров и людей, была сеть UUCP, которая отличалась тем, что у нее абсолютно отсутствовал центральный орган управления. В каком-то смысле это была самая первая пионерская сеть, работавшая через коммутируемые телефонные линии: 9600 bps была какое-то время максимальной скоростью. Самой быстрой сетью (3 Mbps) была сеть, основанная на Ethernet от Xerox и работавшая посредством протокола, названного XNS (Xerox Network Systems). Но она не работала вне локального окружения.

Обстановка того времени довольно сильно отличалась от того, что мы имеем сегодня. Компьютеры были разнотипны до такой степени, что не было даже полного согласия об использовании 8-битных байтов. Например, были компьютеры PDP-10 (36-битные слова, 9-битные байты), PDP-11 (16-битные слова, 8-битные байты), серия CDC 6000 (60-битные слова, 6-битные символы), IBM 360 (32-битные слова, 8-битные байты), XDS 940, ICL 470 и Sigma 7. На подходе была платформа Unix из Bell Laboratories. Большинство Unix-машин использовали 16-битное адресное пространство: в то время PDP-11 была главной машиной на Unix, Data General 8/32 и VAX-11/780 только появлялись. Потоки не существовали – вообще концепция динамических процессов была довольно новой (в Unix они были, но «серезные» системы вроде IBM OS/360 их не имели). Блокировка файлов не поддерживалась ядром Unix (хотя сделать это было возможно используя ссылки файловой системы).

Те сети, которые существовали, были довольно низкоскоростными (многие использовали 9600-бодовые линии TTY; у самых богатых был Ethernet, но только локально). Почтенный интерфейс сокетов еще не будет изобретен в течение многих лет. Шифрование публичными ключами тоже еще не было изобретено, так что большая часть сетевой безопасности, такой как мы ее знаем сегодня, не была возможна.

Сетевая почта уже существовала в Unix, но она была создана через хаки. Основным пользовательским агентом в то время была команда /bin/mail (сегодня иногда называемая как binmail или v7mail), но у некоторых узлов сети были другие агенты, такие как Mail из Беркли, который действительно умел работать с отдельными сообщениями, а не просто был приукрашенной командой cat. Каждый пользовательский агент читал (а обычно и писал!) в /usr/spool/mail напрямую; не было абстракций для настоящего хранения сообщений.

Логика принятия решения о том, направлять ли сообщение в сеть или локально, определялась просто за счет проверки содержания в адресе восклицательного знака (для UUCP) или запятой (для BerkNET). Люди с доступом к Арпанет должны были использовать абсолютно отдельную почто-

вую программу, которая не работала с другими сетями, и которая даже хранила локальную почту в другом месте и другом формате.

Чтобы еще более все запутать, фактически тогда не было никакой стандартизации формата сообщений. Существовало общее соглашение о том, что в начале сообщения должен был быть блок полей заголовка, каждое поле заголовка должно было начинаться с новой строки, имена и значения полей должны были быть разделены запятой. Кроме этого не существовало никакой стандартизации в выборе имен полей заголовка или синтаксисе индивидуальных полей. Например, некоторые системы использовали *Subj:* вместо *Subject:*, поля *Date:* использовали различный синтаксис, некоторые системы не понимали полные имена в поле *From:*

Кроме всего вышеперечисленного, то, что все-таки было задокументировано, зачастую было двусмысленным или не совсем использовалось. В частности, RFC 733 (который должен был описывать формат сообщений Арпанет) отличался от реально используемого формата в небольших, но значимых деталях, а метод непосредственной передачи сообщений вообще не был официально документирован (хотя несколько RFC ссылались на механизм, но не описывали его). Результатом было то, что система передачи сообщений напоминала колдовство.

В 1979 году проект управления реляционными базами данных INGRES (также известные в то время как “моя работа”) получили грант DARPA и вместе с ним 9600 bps подключение нашего PDP-11 к Арпанет. В то время это было единственное доступное соединение к Арпанет в отделе Computer Science, поэтому все хотели получить к нему доступ. Однако наша машина уже была забита и мы могли сделать доступными для всех только два порта. Это приводило к существенным разногласиям и частым конфликтам. Однако, я заметил, что люди обращались больше всего не к удаленному управлению или передаче файлов, а к электронной почте.

Среди всего этого sendmail (изначально названный delivermail) появился как попытка унифицировать хаос в одном месте. Каждый почтовый агент пользователя (или почтовый клиент) просто вызывал delivermail для доставки почты вместо того, чтобы определять как ее доставить самостоятельно, зачастую, произвольным (и непостоянным) образом. Delivermail/sendmail не предпринимали попыток диктовать, как локальная почта должна храниться или доставляться; программа не делала абсолютно ничего, кроме передачи почты между другими программами. (как мы скоро узнаем, эта ситуация изменилась, когда был добавлен SMTP). В каком-то смысле это был просто «клей» для совместной работы различных почтовых систем, вместо того чтобы являться самостоятельной почтовой системой.

Во время разработки sendmail Арпанет была преобразована в Интернет. Изменения не появлялись мгновенно и они были существенными: начиная от низкоуровневых пакетов при подключении до протоколов приложений.

Sendmail был буквально разработан одновременно со стандартами, а в некоторых случаях повлиял на них. Также примечательно то, что sendmail выжил и даже преуспел, несмотря на то, что Сеть (такая, какой мы знаем ее сегодня) выросла с нескольких сотен до миллионов хостов.

Другая сеть

Странно упомянуть, что в то время был предложен еще один, абсолютно отдельный, почтовый стандарт, названный X.400, бывший частью ISO/OSI (International Standards Organization/Open Systems Interconnect). X.400 был бинарным протоколом, сообщения кодировались при помощи ASN.1 (Abstract Syntax Notation 1), который до сих пор используется некоторыми протоколами, например, LDAP. LDAP был свою очередь упрощением X.500, а тот – службой директорий, используемой X.400. Sendmail вообще никак не проектировался для прямой совместимости с X.400, хотя в то время существовали некоторые шлюзовые сервисы. Хотя X.400 был первоначально принят многими коммерческими компаниями того времени, Интернет-почта и SMTP стали стандартом де-факто.

17.2. Принципы разработки

Во время разработки sendmail я придерживался нескольких принципов. Все они в определенном смысле сводились к одному: делать как можно меньше. Это резко контрастирует с некоторыми другими проектами того времени, которые имели гораздо более широкие цели и требовали гораздо более громоздкой реализации.

17.2.1 Один программист ограничен в своих возможностях

Я писал sendmail как побочный неоплачиваемый проект. Мой целью было создать быстрый способ сделать почту Арпанет более доступной для людей в Беркли. Главным было перенаправлять почту между существующими сетями, все они были реализованы как отдельные программы, которые вообще не знали, что существует более, чем одна сеть. Изменение одним не полностью занятым программистом большого количества кода существующих программ было бы невозможно. В процессе разработки необходимо было минимизировать количество кода, который нужно было изменить, и количество вновь написанного кода. Эти ограничения влияли на остальные принципы разработки. Как оказалось, в большинстве случаев, они были правильными, даже если бы была доступна большая команда разработчиков.

17.2.2. Не переделывай пользовательские агенты

Почтовый агент пользователя — это то, что большинство пользователей подразумевают под «почтовой системой» — то есть та программа, которую они используют для чтения почты и ответов на письма. Она сильно отличается от агента передачи почты (MTA), который направляет e-mail от отправителя к получателю. В то время, когда был написан sendmail, многие реализации как минимум частично совмещали эти две функции, поэтому они часто разрабатывались совместно.

Работать над обеими функциями сразу было бы слишком тяжело, поэтому с sendmail я полностью отказался от идеи переделки пользовательского интерфейса: единственное изменения в пользовательских агентах были в том, чтобы они вызывали sendmail вместо выполнения собственной маршрутизации. Кроме того тогда уже было несколько пользовательских агентов, и люди стали привыкать к тому, как они работают с почтой. Пытаться работать над всеми функциями сразу не представлялось возможным. Такое разделение пользовательского агента от агента передачи почты естественно сейчас, но тогда это сильно отличалось от общепринятых решений.

17.2.3. Не переделывай хранилище локальной почты

Хранилище локальной почты (место, где должны сохраняться сообщения до тех пор, пока получатель их не прочтет) не было формально стандартизовано. Некоторые узлы предпочитали хранить ее в централизованном месте, таком как /usr/mail, /var/mail или /var/spool/mail. Другие хранили почту в домашней папке получателя (как файл .mail). Большая часть узлов начинала каждую строку с “From” и затем шел пробел (чрезвычайно неудачное решение, но так было принято в то время), но те узлы, которые нацеливались на Арпанет, обычно хранили сообщения, разделяя их строкой, содержащей четыре control-A символа.

Некоторые узлы пытались блокировать почтовый ящик для предотвращения коллизий, но они использовали разные договоренности о блокировке (примитивы блокирования файлов еще не были доступны). Короче говоря, единственной разумной вещью было считать хранилище локальной почты черным ящиком.

Практически на всех узлах механизм работы с хранилищем локальной почты был внедрен в программу /bin/mail. Она имела (довольно примитивный) пользовательский интерфейс, маршрутизацию и хранение, реализованные в одной программе. Для подключения sendmail часть, отвечающая за маршрутизацию, была вынута и заменена на вызов sendmail. Был добавлен флаг -d для возмож-

ности явного указания необходимости конечной доставки, другими словами он не давал вызывать sendmail из /bin/mail. В последующие годы код, используемый для доставки сообщения в физический почтовый ящик, был выделен в отдельную программу mail.local. Программа /bin/mail существует сегодня только как место вызова отправки почты, сама эти функции не выполняет.

17.2.4. Пусть sendmail подстраивается под мир, а не наоборот

Такие протоколы как UUCP и BerkNET уже были реализованы как отдельные программы, имеющие свои собственные, иногда мудреные, интерфейсы командой строки. В некоторых случаях они активно развивались одновременно с sendmail. Было понятно, что пытаться сделать иную их реализацию (например, для приведения к стандартным способам вызова) было бы неудобно. Это напрямую приводило к принципу, что sendmail должен адаптироваться к остальному миру, а не пытаться подстроить остальные программы под sendmail.

17.2.5. Менять как можно меньше

В максимально возможной степени во время разработки sendmail я не трогал то, что можно было не трогать. Кроме того, что на это просто не было времени, в то время в Беркли вместо более формальных правил определения принадлежности кода склонялись в пользу правила «кто последний трогал код, к тому и будем обращаться по поводу этой программы» (или проще говоря «тронул код, теперь он твой»). Хотя это звучит довольно беспорядочно по сегодняшним меркам, это работало в то время в Беркли, когда никто не трудился полный рабочий день над Unix; каждый работал над частью системы, которая им была интересна, и не трогал остальной код за исключением обстоятельств крайней необходимости.

17.2.6. Сразу думай о надежности

Почтовые системы до sendmail (включая большинство транспортных систем) не очень были озабочены надежностью. Например, версии Unix до 4.2BSD не имели встроенной поддержки блокировки файлов, хотя ее можно было симулировать, создав временный файл и затем связав его с файлом блокировки (если файл блокировки уже существовал, вызов link не работал). Однако, иногда различные программы, писавшие данные в один и тот же файл, не договаривались о том, как должна выполняться блокировка (например, они могли использовать различные имена для файлов блокировки или вообще не пытаться блокировать файл), и поэтому не было из ряда вон выходящим, если почта терялась. Sendmail занял другую позицию: терять почту было нельзя (возможно, как результат моего опыта работами с базами данных, где потеря данных является смертным грехом).

17.2.7. Что не было реализовано

Многие вещи не были реализованы в ранних версиях. Я не пытался переделать архитектуру почтовой системы или создать полноценное общее решение: функциональность могла быть добавлена по мере необходимости. Самые ранние версии не были даже полностью конфигурируемыми без доступа к исходным кодам и компилятору (хотя это было довольно быстро исправлено). В общем, modus operandi для sendmail было сделать как можно раньше работающую версию и затем улучшать работающий код по необходимости, когда проблема будет более ясна.

17.3. Фазы разработки

Как и у большинства долгоживущих проектов у sendmail было несколько этапов разработки, каждый со своей общей целью и ощущениями.

17.3.1. Волна первая: delivermail

Первая реализация sendmail была известна как delivermail. Она была чрезвычайно простой, если не сказать примитивной. Ее единственной задачей было перенаправлять сообщения от одной программы к другой; в частности, у нее не было поддержки SMTP, и она никогда не делала никаких прямых сетевых соединений. Не было необходимости в очереди, потому что каждая сеть имела свою, поэтому программа по сути была просто координатным коммутатором. Так как у delivermail не было прямой поддержки сетевых протоколов, не было причин запускать ее как демон – она вызывалась для перенаправления каждого сообщения, когда то отправлялось. Программа передавала сообщение подходящей программе, которая реализовывала следующий этап и завершалась.

Не было попыток переписывать заголовки для соответствия сети, в которую доставлялось сообщение. Это обычно приводило к тому, что на перенаправляемые сообщения нельзя было ответить. Ситуация была настолько плохой, что об адресации писем была написана целая книга (названная соответствующе !%@:: Руководство по адресации электронной почты и сетям [AF94]).

Вся конфигурация в delivermail была внутри исходного кода и основывалась на специальных символах в адресах. Символы имели приоритет. К примеру, конфигурация хоста могла искать знак “@” и, если таковой был найден, посыпала весь адрес назначенному транслирующему хосту Арпанет. Если его не было, она могла искать запятую, и затем посыпать сообщение назначенному хосту BerkNET и пользователю, если такой был найден, затем могла проверять наличие восклицательного знака (“!”), сигнализирующего о том, что сообщение должно быть передано ретранслятору UUCP. Если и такого не было, совершалась попытка локальной доставки. Эта конфигурация могла выглядеть вот так:

```
Input Sent To {сеть, хост, пользователь}
foo@bar      {Arpanet, bar, foo}
foo:bar      {Berknet, foo, bar}
foo!bar!baz  {Uucp, foo, bar!baz}
foo!bar@baz  {Arpanet, baz, foo!bar}
```

Обратите внимание, что разделители в адресе указывались по-разному, что приводило к путанице, которая могла быть исправлена только при помощи эвристических методов. Например, последний из приведенных примеров мог вполне быть распарсен как {Uucp, foo, bar@baz} на одном из узлов.

Конфигурация была внедрена в исходный код по нескольким причинам: во-первых, из-за ограниченной памяти и 16-битного адресного пространства, парсинг конфигурации во время выполнения программы был бы слишком затратным. Во-вторых, системы того времени были настолько сильно заточены под собственные задачи их владельцев, что перекомпиляция была сама по себе хорошей идеей, просто чтобы убедиться, что у вас есть локальные версии используемых в них библиотек (совместно используемые библиотеки не существовали в Unix 6).

Delivermail распространялась с 4.0 и 4.1 версиями BSD и была более успешной, чем ожидалось; Беркли был далеко не единственным узлом с гибридной сетевой архитектурой. Стало ясно, что требовалась дополнительная работа над этим проектом.

17.3.2. Волна 2: sendmail 3, 4 и 5

Версии 1 и 2 распространялись под названием delivermail. В марте 1981 началась работа над версией 3, которая должна была выйти под именем sendmail. В это время 16-битная PDP-11 все еще была распространена, но 32-битная VAX-11 становилась все популярнее, так что многие имевшиеся изначально ограничения, связанные с малым пространством адресов, отходили в прошлое.

Первоочередными задачами для sendmail были переход на конфигурирование во время выполнения, разрешение модификаций сообщений для обеспечения совместимости между сетями для перенаправляемой почты, а также более богатый язык, на основе которого будут приниматься решения о маршрутизации. Для этого по существу использовалась текстовая перезапись адресов (основанная на токенах, а не символьных строках), этот механизм использовался в некоторых эксперта-

ных системах того времени. Существовал специально написанный для этого код для извлечения и сохранения всех строк комментариев (в скобках) и последующей их вставки после того, как программная перезапись была завершена. Было также важно иметь возможность добавлять или пополнять поля заголовков (например, добавлять заголовок Date или включать полное имя отправителя в заголовок From, если оно было известно).

Разработка SMTP началась в ноябре 1981. Исследовательская группа компьютерных наук в университете Беркли получила контракт DARPA на разработку платформы, основанной на Unix для поддержки исследований, спонсируемых DARPA, с целью сделать совместный доступ между проектами проще.

Первоначальная работа над стэком TCP/IP была к тому времени завершена, хотя детали интерфейса сокетов еще модифицировались. Основные протоколы приложений, такие как Telnet и FTP, были завершены, но SMTP еще не был реализован. На самом деле протокол SMTP еще даже не был окончен на тот момент; были большие дискуссии на тему того, что почта должна была пересыпаться при помощи протокола, креативно названного Протокол передачи почты (Mail Transfer Protocol – MTP). В то время, как дебаты разгорались, MTP становился все более и более сложным до тех пор, пока от разочарования в нем не был создан набросок протокола SMTP (Простой Протокол Передачи Почты – Simple Mail Transfer Protocol – SMTP), который не был официально опубликован до августа 1982 года. Официально я работал на INGRES Relational Database Management System, но так как я знал о почтовых системах больше, чем кто-либо в Беркли в то время, со мной стали говорить о реализации SMTP.

Оей первой мыслью было создать отдельного SMTP-отправителя, который имел бы свою собственную организацию очередей и демон; эта подсистема подсоединялась бы к sendmail для маршрутизации. Однако, некоторые особенности SMTP делали это проблематичным. Например, команды EXPN и VRFY требовали доступа к парсингу, созданию алиасов и модулю проверки локальных адресов. Также, в то время я считал важным, чтобы команда RCPT возвращала результат мгновенно, если адрес был не известен, а не принимала сообщение и затем отправляла бы сообщение о неудавшейся доставке позже. Это оказалось очень дальновидным решением. Забавно, что позже агенты передачи почты часто реализовывали этот момент неправильно, усугубляя проблему обратной отправки спама. Эти проблемы привели к решению сделать SMTP частью sendmail.

Sendmail 3 распространялась с версиями 4.1a и 4.1c BSD (бета версии), sendmail 4 распространялся с версией 4.2 BSD, а sendmail 5 – с версией 4.3 BSD.

17.3.3. Волна 3: Годы хаоса

После того, как я покинул Беркли и перешел в стартап, время на работу над sendmail резко сократилось. Но интернет начинал серьезно набирать обороты и sendmail использовался в различном новом, и все более масштабном, окружении. Большинство производителей Unix-систем (Sun, DEC и IBM) в частности создали свои собственные версии sendmail, взаимно не совместимые между собой. Были также попытки создать opensource-версии, в частности IDA sendmail и KJS.

IDA sendmail появилась из университета Linköping University. IDA включала расширения для более легкой установки и работы в масштабируемых системах, а также абсолютно новую систему конфигурации. Одной из главных новых функций было включение базы данных dbm(3) для поддержки узлов с динамическим содержимым. Поддержка базы данных была возможна за счет нового синтаксиса конфигурационного файла и использовались для многих функций, включая преобразование адресов в/из внешнего синтаксиса (например, отправка письма на ящик john_doe@example.com вместо johnd@example.com) и для маршрутизации.

Sendmail короля Джеймса (KJS, разработана Paul Vixie) была попыткой объединения всех различных версий sendmail. К сожалению, она так и не получила достаточной поддержки для достижения

желаемого результата. Эта эра была также обозначена множеством новых технологий, которые отразились на почтовой системе.

Например, создание Sun бездисковых кластеров добавило службу директорий YP (позже NIS) и NFS, сетевую файловую систему. В частности YP должна была быть видимой для sendmail, т.к. алиасы хранились в YP, а не в локальных файлах.

17.3.4 Волна 4: sendmail 8

Спустя несколько лет я вернулся в Беркли как штатный сотрудник. Моей работой было управление группой, занимавшейся установкой и поддержкой инфраструктуры с совместным доступом для исследований отдела Вычислительной техники. Чтобы решать поставленные задачи, созданные главным образом спонтанно исследовательские группы должны были быть объединены каким-то рациональным образом. Во многом как и в ранние дни Интернета, различные исследовательские группы работали на абсолютно различных платформах, некоторые из них были довольно устаревшими. В общем, каждая группа имела свои системы, и хотя некоторые из них были хорошо управляемы, большинство страдало от «отложенного ремонта».

В большинстве случаев электронная почта также была не структурирована. У каждого был адрес электронной почты “`person@host.berkeley.edu`”, где хостом было название рабочей станции в их офисе или используемый ими общий сервер (университет не имел внутренних поддоменов), за исключением нескольких особенных людей, которые имели адрес `@berkeley.edu`. Задачей было перейти на внутренние поддомены (так чтобы все индивидуальные хосты имели поддомен `cs.berkeley.edu`) и объединенную почтовую систему (чтобы каждый человек имел адрес `@cs.berkeley.edu`). Эту задачу легче всего было решить, создав новую версию sendmail, которую можно было бы использовать во всем отделе.

Я начал с изучения различных вариантов sendmail, получивших популярность. В мои намерения не входило начинать с абсолютно нового кода, а скорее понять функциональность, которую другие нашли для себя полезной. Многие из тех идей были реализованы в sendmail 8, часто с модификациями для совмещения со схожими идеями или для того, чтобы представить их в более общем виде. Например, несколько версий sendmail имели возможность доступа к внешним базам данных, таких как dbm(3) или NIS; sendmail 8 объединил их в один механизм преобразования, который мог работать со многими базами данных (и даже с преобразованием произвольных данных не из баз данных). Аналогично, была включена обобщенная работа с базами данных (сопоставление внутренних и внешних имен) из IDA sendmail.

Sendmail 8 также включал новый конфигурационный пакет, использующий m4(1) макропроцессор. Он был более декларативным, чем конфигурационный пакет sendmail 5 (главным образом процедурный). То есть sendmail 5 требовал от администратора полностью вручную создавать конфигурационный файл, используя только возможность “`include`” из m4 для удобства. Конфигурационный файл sendmail 8 позволял администратору задавать, какие функции, отправители сообщений и так далее требовались, а m4 выдавал конечный конфигурационный файл.

Большая часть секции 17.7 описывает улучшения в sendmail 8.

17.3.5 Волна 5: Коммерческие годы

С ростом Интернета и увеличения количества серверов, использовавших sendmail, поддержка больших баз пользователей становилась все более проблематичной. Какое-то время я мог продолжать этот процесс, создав группу добровольцев (неформально названную “Консорциум Sendmail”, или `sendmail.org`), которые обеспечивали бесплатную поддержку по e-mail и через группу новостей. Но к концу 1990-х количество пользователей выросло настолько, что их практически невозможно было поддерживать на добровольной основе. Вместе с более бизнес-ориентированным другом я создал Sendmail, Inc., ожидая получить новые ресурсы для работы над кодом.

Хотя коммерческий продукт был изначально главным образом ориентирован на конфигурационные инструменты, многие новые функции были добавлены в агент передачи посты с открытым исходным кодом для поддержки нужд коммерческого мира. В частности, компания добавила поддержку TLS (шифрование соединения), SMTP-аутентификации, улучшения в безопасности узлов, такие как защиту от отказа в обслуживании, и, самое важное, плагины для фильтрации почты (интерфейс Milter'а описывается ниже).

На момент написания этой книги коммерческий продукт разросся настолько, что стал включать большие пакеты приложений, основанных на e-mail, большинство которых были созданы на основе расширений, добавленных в sendmail во время первых лет существования компании.

17.3.6. Что произошло с sendmail 6 и 7?

Sendmail 6 по сути был бета-версией sendmail 8. Эта версия никогда не была официально выпущена, но получила довольно широкое распространение. Sendmail 7 вообще никогда не существовало; сразу была выпущена версия 8, потому что все остальные файлы исходных кодов для дистрибутива BSD были выпущены как версия 8, когда BSD 4.4 была выпущена в июне 1993.

17.4. Проектные решения

Некоторые решения в процессе проектирования были правильными. Некоторые начинались правильно, но стали неверными, так как мир успел измениться. Некоторые были сомнительными и так и не стали менее сомнительными со временем.

17.4.1. Синтаксис конфигурационного файла

Синтаксис конфигурационного файла преследовало несколько проблем. Во-первых, все приложение должно было помещаться в 16-битном адресном пространстве, поэтому парсер должен был быть маленьким. Во-вторых, первые конфигурации были довольно небольшими по размеру (до одной страницы), поэтому несмотря на запутанный синтаксис, файл все равно оставался читаемым. Однако по прошествии времени все больше решений по работе программы переносились из кода C в конфигурационный файл, и файл начал разрастаться. Конфигурационный файл приобрел репутацию загадочного. Одной из проблем для многих был выбор символа табуляции в качестве активного синтаксического элемента. Это была одна из ошибок, скопированная с других систем того времени, в частности make. Эта ошибка стала более критичной с появлением оконных систем (а с ними копирования и вставки, которые обычно не сохраняют табуляцию).

Оглядываясь назад, я понимаю, что так как файл становился все больше и начали появляться 32-битные машины, имело смысл пересмотреть синтаксис. Было время, когда я раздумывал над этим, но решил не делать, т.к. не хотел изменять конфигурации на «большом» количестве машин (которое в то время составляло несколько сотен). Как оказалось это было ошибкой; я просто недооценил насколько вырастет количество инсталляций программы и сколько часов я бы сэкономил, если бы поменял синтаксис раньше. Также, когда стандарты устоялись, большое количество опций можно было вернуть в код программы, упростив ее настройку.

Отдельный интерес представляет то, насколько много функциональности добавилось в конфигурационный файл. Я разрабатывал sendmail одновременно с развитием стандарта SMTP. За счет перемещения функциональных решений в конфигурационный файл я мог быстро отвечать на изменения в стандарте – обычно в течение 24 часов. Я думаю, что это улучшило SMTP-стандарт, так как представлялось возможным получить опыт работы с предлагаемыми изменениями достаточно быстро, хотя ценой этого стал трудночитаемый конфигурационный файл.

17.4.2. Правила преобразования (Rewriting Rules)

Одним из трудных вопросов при написании sendmail было то, как делать необходимые преобразования для разрешения пересылки между сетями без нарушения стандартов получающей сети. Требовалось изменения метасимволов (например, BerkNET использовала запятую в качестве разделителя, которую нельзя было применять в адресах в SMTP), перемещения компонентов адресов, добавления или удаления компонентов и так далее. Например, следующие преобразования были необходимы в определенных обстоятельствах:

Из	В
a:foo	a.foo@berkeley.edu
a!b!c	b!c@a.uucp
<@a.net,@b.org:user@c.com>	<@b.org:user@c.com>

Регулярные выражения не были лучшим выбором, потому что они не имели хорошей поддержки для границ слов, кавычек и т.д. Быстро стало ясно, что практически невозможно написать регулярные выражения, которые были бы точными, и уж тем более вразумительными. В частности регулярные выражения резервировали несколько метасимволов, включая “.”, “*”, “+”, “[{}]” и “[{}]{}”, а все они могли появляться в e-mail адресах.

Эти символы могли быть экранированы в конфигурационных файлах, но я считал это сложным, запутывающим и достаточно уродливым (такое решение попробовали в UPAS в Bell Laboratories, мейлер из Unix 8, но он никогда не завоевал популярность). Вместо этого, был необходим этап сканирования для создания токенов, которыми затем можно было манипулировать также как символами в регулярных выражениях. Единственного параметра, описывающего «управляющие символы», которые сами были и токенами, и разделителями токенов, было достаточно. Пустые пробелы разделяли токены, но сами токенами не были. Правила преобразования были просто паттернами сопоставления и замены пар, организованные по сути в подпрограммы.

Вместо большого количества метасимволов, которые нужно было экранировать, чтобы убрать их «магические» свойства (как они использовались в регулярных выражениях), я использовал один единственный «экранирующий» символ, который комбинировался с обычными символами для представления паттернов с произвольным символом (например, для поиска произвольного слова).

Традиционным подходом Unix было использование обратного слэша, но обратный слэш уже использовался как символ кавычек в некоторых адресах. Как оказалось, “\$” был одним из немногих символов, которые еще не использовались как пунктуационные символы в синтаксисе email.

Одним из первональных неудачных решений было, как ни странно, то, как использовались пробелы. Символ пробела был разделителем, как и в большинстве вводимых с компьютера данных, и поэтому мог быть использован свободно между токенами в паттернах. Однако, первые распространяемые конфигурационные файлы не включали пробелы, что приводило к паттернам, которые было понять труднее, чем это было необходимо. Посмотрите на разницу между следующими двумя (семантически идентичными) паттернами:

```
$+ + $* @ $+ . $={mydomain}
$+$*@$+. ${mydomain}
```

17.4.3. Использование преобразования для парсинга

Некоторые предлагали, что sendmail должен использовать для разбора адресов стандартные техники парсинга, основанные на грамматике, а не правила преобразования и оставить их для модификации адресов. На первый взгляд это имело смысл, учитывая, что стандарты, определяющие адреса, используют грамматику. Главной причиной повторного использования правил преобразования было то, что в некоторых случаях было необходимо разбирать адреса поля заголовка (например, чтобы выбрать конверт отправителя из заголовка при получении почты из сети, не име-

шей формального конверта). Такие адреса нелегко распарсить, используя, скажем, парсер LALR(1) вроде YACC и традиционный сканер, из-за количества символов, которые необходимо просмотреть впереди.

Например, разбор адреса `allman@foo.bar.baz.com <eric@example.com>` требует просмотра вперед или сканером или парсером; вы не можете знать что первоначальный “`allman@...`” не является адресом до тех пор, пока не увидите “`<`”. Так как парсеры LALR(1) имеют только один токен для просмотра вперед, это будет необходимо делать в сканере, что заметно его усложнит. По той причине, что правила преобразования уже поддерживали произвольный перебор с возвратами (то есть они могли смотреть вперед произвольно далеко), их было достаточно.

Второй причиной было то, что через паттерны было относительно легко распознать и исправить неверный ввод. В конце концов, преобразования через правила были более чем мощными для данной задачи, а повторное использование кода было мудрым решением.

Один необычный момент о правилах преобразования: при сопоставлении с паттерном полезно и для ввода и для паттерна создать токены. По этой причине один и тот же сканер был использован для вводимых адресов и для самих паттернов преобразования. Для этого сканер вызывался с различными таблицами типов символов для различных вводимых данных.

17.4.4. Внедрение SMTP и организации очереди в sendmail

«Очевидным» способом реализации исходящей (клиентской) части SMTP было бы сделать его как внешний отправитель, схожий с UUCP, но это привело бы к ряду вопросов. Например, где нужно было делать организацию очередей – в `sendmail` или другом клиентском SMTP-модуле? Если делать ее в `sendmail`, тогда отдельные копии сообщений нужно было рассыпать каждому получателю (т.е. нельзя было бы открыть одно соединение и затем рассыпать несколько RCPT-команд) или гораздо более сложный способ обратной коммуникации был бы необходим для передачи необходимого состояния каждого получателя, чем это было возможно с использованием простых кодов завершения Unix.

Если организация очереди была сделана в клиентском модуле, то был большой потенциал для репликации; в частности в то время другие сети, например, XNS еще были возможными соперниками. Кроме того, включение очереди в сам `sendmail` предоставляла более элегантный способ работы с определенного рода ошибками, в частности кратковременными проблемами, такими как нехватка ресурсов.

Входящая (серверная) часть SMTP подразумевала другой набор решений. В то время я считал важным реализовать VRFY и EXPN SMTP-команды точно, что требовало доступа к механизму алиасов. Это опять же потребовало бы гораздо более сложного протокола обмена между серверным модулем SMTP и `sendmail`, чем было возможно при использовании командных строк и кодов завершения (на самом деле, что-то вроде самого протокола SMTP для такой коммуникации).

Сегодня я бы гораздо более склонялся к тому, чтобы оставить реализацию очереди в ядре `sendmail`, но переместить обе части реализации SMTP в другие процессы. Одна из причин для этого в безопасности: как только на стороне сервера открыт 25 порт, нет необходимости в доступе администратора. Современные расширения, такие как TLS и DKIM-подписьание, усложняют клиентскую сторону (т.к. секретные ключи не должны быть доступны непrivилегированным пользователям), но строго говоря доступ под администратором все равно так и не нужен. Несмотря на то, что если SMTP-клиент работает как пользователь-неадминистратор, имеющий возможность на чтение секретных ключей, проблема безопасности остается, у этого пользователя по определению есть особые привилегии, и поэтому он не должен напрямую осуществлять коммуникации с другими узлами. Все эти проблемы можно обойти довольно просто.

17.4.5. Реализация очереди

Sendmail следовал представлениям того времени о хранении файлов очереди. На самом деле используемый формат был чрезвычайно похож на подсистему lpr (линейного печатающего принтера) того времени. Каждое задание имело два файла, один с контрольной информацией и один с данными. Контрольный файл был простым текстовым файлом, первый символ каждой строки содержал информацию о значении этой строки.

Когда sendmail хотел обработать очередь, ему нужно было прочитать все контрольные файлы, сохранив всю нужную информацию в память, и затем сортировать этот список. Это хорошо работало для относительно маленького количества сообщений, но при числе сообщений в очереди больше 10 000 не давало нужный результат. В частности, когда папка становилась настолько большой, что требовала блоков с непрямым доступом в файловой системе, возникала серьезная проблема с производительностью, которая могла снизить быстродействие на порядок. Было возможно улучшить данную ситуацию, заставив sendmail понимать несколько директорий с данными очередей, но это был в лучшем случае хак.

Альтернативным подходом было хранение всех контрольных файлов в базе данных. Это не было сделано, потому что в то время не было доступных пакетов баз данных, а когда появилась dbm(3), у нее было несколько недостатков, включая невозможность возвращения места на диске, требование о том, что все ключи, хэшируемые вместе, должны помещаться на одной странице (512 байт) и отсутствие блокировки. Надежные пакеты баз данных не появлялись еще многие годы.

Еще одной возможностью альтернативной реализации было создание отдельного демона, хранящего состояние очереди в памяти, возможно, записывающего также информацию в лог для возможности восстановления. Но по причине относительно небольшого объема почтового траффика в то время, недостатка памяти на большинстве машин, относительно высоких затрат на фоновые процессы и сложность реализации такого процесса, это решение вряд ли было хорошим в то время.

Еще одним проектировочным решением было хранение заголовков сообщения в контролльном файле очереди, а не в файле данных. Объяснением этому было то, что большинство заголовков требовали существенных изменений, зависящих от места назначения (а так как сообщения могли иметь более одного места назначения, их нужно было изменять несколько раз), и цена разбора заголовков оказывалась достаточно высокой, поэтому хранение их в формате, предварительно подготовленном для разбора, казалось экономией.

В ретроспективе это оказалось не лучшим решением, также как и хранение тела сообщения в стандартном формате Unix (с завершающими символами начала строки), а не в формате, в котором они были получены (который мог использовать символы начала новой строки, возврата каретки/перевода строки, просто возврата каретки или перевода строки/возврата каретки). С развитием мира электронной почты и принятием стандартов, необходимость в перезаписи сократилась, а даже кажущееся безобидным преобразование содержит риск ошибки.

17.4.6. Получение и исправление неверного ввода

Так как sendmail был создан в мире множества протоколов и на редкость малого количества стандартов, я решил приводить в порядок плохо сформированные сообщения, когда это возможно. Это соответствует “принципу устойчивости” (он же закон Постеля), сформулированному в RFC 7934. Некоторые из этих изменений были очевидными и даже обязательными: при отправке UUCP сообщения в Арпанет адреса UUCP нужно было конвертировать в адреса Арпанет, чтобы команда “Ответить” работала корректно, завершающие строки символы нужно было конвертировать между форматами, используемыми различными платформами, и так далее. Некоторые были менее очевидными: если сообщение было получено, но не включало заголовок “From:”, которое было обязательным согласно спецификациям Интернета, нужно ли было добавлять это поле заголовка, передавать сообщение без него или отказать в отправке?

В то время моей главной заботой была функциональная совместимость, поэтому sendmail исправлял сообщение, например, добавляя поле заголовка `From:`. Однако говорили о том, что это позволяло другим старым почтовым системам существовать еще долгое время после того, как они уже должны были быть исправлены или от них нужно было вовсе отказаться.

Я думаю, что мое решение было верным для того времени, но сегодня является причиной проблем. Высокая степень функциональной совместимости была важна, чтобы поток писем отправлялся без препятствий. Если бы я отклонял неверно сформированные сообщения, большинство сообщений в то время были бы отклонены. Если бы я пропускал их неисправленными, получатели бы получали сообщения, на которые не могли бы ответить и в некоторых случаях – сообщения, по которым нельзя было бы определить, кем они были отправлены, или же сообщения были бы отклонены другим почтовым отправителем.

Сегодня существуют стандарты, и по большей части эти стандарты точные и полные. Больше нет проблем в том, что многие сообщения будут отклонены. И тем не менее существует почтовое программное обеспечение, которое рассыпает некорректно сформированные почтовые сообщения. Это создает многочисленные ненужные проблемы для других программ в Интернете.

17.4.7. Конфигурирование и использование M4

Какое-то время я одновременно регулярно вносил изменения в конфигурационные файлы sendmail и лично поддерживал многие машины.

Так как большое количество конфигурационных файлов на разных машинах оставалось одинаковым, становилось желательно использовать какой-то инструмент для создания конфигурационных файлов. Макропроцессор `m4` был включен в Unix. Он был разработан как клиентская часть программных языков (в частности, `ratfor`). Что важнее, у него были возможности “`include`”, как инструкция “`#include`” в языке C. Первоначальные конфигурационные файлы использовали немногим больше этой возможности и небольшие макро-расширения.

IDA sendmail также использовала `m4`, но абсолютно другим образом. Оглядываясь назад, я думаю, что мне нужно было изучать эти образцы более детально. В них содержалось много умных идей, в частности то, как обрабатывалось экранирование.

Начиная с sendmail 6 конфигурационные файлы `m4` были полностью переписаны в более декларативном стиле и стали гораздо меньше. Они использовали гораздо больше возможностей процессора `m4`, что привело к некоторым проблемам, когда появление GNU `m4` изменило некоторую часть семантики в небольшой степени.

Изначально замысел был в том, что конфигурации `m4` должны были следовать правилу 80/20: они должны были быть простыми (отсюда 20% работы) и должны покрывать 80% случаев. Довольно быстро стало понятно, что этого не получилось, по двум причинам.

Не самой важной причиной оказалось то, что было довольно легко конфигурировать большую часть случаев, по крайней мере по началу. Но это стало гораздо сложнее по мере развития sendmail и мира, особенно при включении таких функций как TLS-шифрования и SMTP-аутентификации.

Важная причина была в том, что использовать обычные конфигурационные файлы было слишком сложным для большинства людей. По сути формат файлов `.cf` (необработанный) стал компонующим автокодом – в принципе редактируемым, но в реальности довольно запутанным. “Исходный код” представлял собой `m4` скрипт в файлах `.mc`.

Еще одним важным различием было то, что конфигурационный файл в необработанном виде представлял собой язык программирования. В нем были процедуры (наборы правил), вызовы под-

программ, детализация параметров и циклы (но без операторов перехода `goto`). Синтаксис был запутанным, но во многих случаях напоминал команды `sed` и `awk`, по крайней мере по сути.

Формат `m4` был декларативным: хотя было возможно использовать низкоуровневый язык, на практике эти детали были скрыты от пользователя.

Неясно, было ли это правильным или неправильным решением. Я считал в то время (и до сих пор считаю), что при создании сложных систем можно сделать то, что называется специализированным языком (Domain Specific Language – DSL) для создания определенных разделов этой системы. Однако, использование такого языка как методологии конфигурирования для конечного пользователя по сути превращает все попытки конфигурирования этой системы в проблему программирования. В этом заключены большие возможности, но и цена этого также высока.

17.5. Другие соображения

Некоторые другие проектировочные и архитектурные моменты, которые стоит отметить.

17.5.1. Об оптимизации масштабных Интернет-систем

В большинстве сетевых систем существуют конфликты между клиентом и сервером. Хорошая стратегия для клиента может быть неверной для сервера и наоборот. Например, при возможности на сервере стараются минимизировать затраты на обработку данных, перекладывая эти обязанности как можно больше на клиента, и конечно клиенту также необходимы подобные меры, но в обратном направлении.

Например, на сервере может быть выгоднее держать соединение открытым, обрабатывая спамсообщения, так как это снижает затраты на отклонение сообщений (что на сегодняшний день является обычным делом), но клиент хочет отключиться как можно быстрее. Если посмотреть на всю систему целиком, Интернет в целом, оптимальным решением может быть баланс между этими двумя необходимостями.

Существовали такие агенты передачи почты, которые явно предпочитали один или другой способ – клиента или сервера. Они могут себе это позволить только потому, что у них довольно немного установок. Если ваша система используется на значительной части Интернета, вам необходимо спроектировать ее так, чтобы сбалансировать нагрузку между двумя сторонами в попытках оптимизировать Интернет как целое. Эта задача осложняется тем, что всегда будут существовать агенты передачи, полностью использующие только один подход, например, массовые почтовые системы, заботящиеся только об оптимизации исходящей стороны.

При разработке системы, которая включает обе стороны соединения, важно избегать подобных предпочтений. Заметьте, что это находится в разительном контрасте с обычной асимметрией клиентов и сервисов, например, веб-серверы и веб-клиенты обычно не разрабатываются одной и той же группой.

17.5.2. Milter

Одним из наиболее важных дополнений к `sendmail` был интерфейс `milter` (mail filter). `Milter` позволяет использование сторонних плагинов для обработки почты. Изначально они разрабатывались для антиспамовой обработки. Протокол `milter` работает синхронно с протоколом сервера SMTP. По мере того, как каждая новая команда SMTP, получается от клиента, `sendmail` вызывает `milter` с информацией от этой команды. `Milter` имеет возможность принять эту команду или послать отказ, что отклоняет фазу протокола, соответствующую команде SMTP. Фильтры `Milter` созданы как callback-функции, поэтому когда приходит команда SMTP, вызывается соответствующая подпро-

грамма milter. Фильтры Milter работают как подпроцессы, указатель контекста передается в каждом соединении для каждой подпрограммы, что позволяет передачу текущего состояния.

В теории фильтры milter могли работать как подгружаемые модули в пространстве адресов sendmail. Мы отклонили это решение по трем причинам. Во-первых, вопросы безопасности были слишком значительны: даже если sendmail был запущен под уникальным пользователем-неадминистратором, этот пользователь имел бы доступ ко всем состояниям всех остальных сообщений. Было неизбежно, что некоторые авторы фильтров milter попытались бы получить доступ к внутреннему состоянию sendmail. Во-вторых, мы хотели создать экран между sendmail и фильтрами milter: если фильтр падал, мы хотели, чтобы было ясно, чья была ошибка, и при этом потенциально почта продолжала отправляться. В-третьих, автору фильтра было гораздо легче отлаживать отдельный процесс, а не весь sendmail. Быстро стало понятно, что фильтры были полезны не только для обработки спама. Сайт milter.org содержит список фильтров для борьбы со спамом, вирусами, архивации, мониторинга контента, логирования, изменения трафика и многих других категорий. Фильтры создаются коммерческими компаниями и в качестве проектов с открытым исходным кодом. Программа postfix также добавила поддержку фильтров milter, используя тот же самый интерфейс. Фильтры Milter оказались одним из самых успешных введений sendmail.

17.5.3. Расписание релизов

Довольно популярны споры между двумя подходами – “выпускай быстро и часто” и “выпускай стабильные системы”. Sendmail использовал и тот, и другой подход в разное время. Во время значительных изменений иногда я делал более, чем один релиз в день. Моеей общей философией было делать релиз после каждого изменения. Это похоже на предоставление публичного доступа к системному дереву управления исходным кодом. Лично я предпочитаю делать релизы, а не давать публичный доступ к дереву исходного кода, отчасти потому что я использую управление исходным кодом таким образом, который сейчас не одобряется: при больших изменениях я заношу в систему управления контроля версий нефункционирующие версии кода. Если дерево открыто для общего доступа, я использую для этих версий ветки (branches), но в любом случае они доступны для всех и это может привести к значительной путанице. Кроме того, создание релиза означает задание для него номера, а это облегчает отслеживание изменений при работе с баг-репортами. Конечно, это требует того, чтобы релизы можно было делать легко, а это не всегда так.

Со временем, когда sendmail начал использоваться во все более критичном окружении, это стало вызывать проблемы. Другим не всегда было легко понять, было ли выпущенное изменение для тестирования или оно действительно могло использоваться в продакшне. Наименование релизов как “альфа” или “бета” помогало, но не решало проблему. Результатом стало то, что с развитием sendmail перешел от более частых к более объемным релизам. Это стало особенно критично, когда sendmail стал выпускаться коммерческой компанией, клиенты которой хотели последние и самые лучшие и при этом одновременно и стабильные версии и не приняли бы того, что релизы нестабильны.

Эти разногласия между нуждами open source разработчиков и коммерческим продуктом никогда не исчезнут. Есть множество преимуществ в ранних и частых релизах, в частности потенциально большая аудитория отважных (и иногда глуповатых) тестировщиков, которая проверяет систему способами, которые вы никогда не ожидали бы повторить в стандартной системе разработки. Но как только проект становится успешным, есть тенденция превратить его в продукт (даже если этот продукт с открытыми исходными кодами и бесплатный), а продукты имеют другие потребности, нежели проекты.

17.6. Безопасность

В том, что касается безопасности, у sendmail была беспокойная жизнь. Некоторая часть из плохой славы была заслужена, но некоторая нет, так как концепция “безопасности” менялась на наших глазах.

Интернет начинался как база пользователей размером в несколько тысяч человек, главным образом из академических и исследовательских кругов. Во многом это был более добрый и мягкий интернет, нежели тот, что мы знаем сегодня. Сеть была спроектирована таким образом, чтобы поощрять распространение материалов, а не создание фаерволов (эта концепция вообще не существовала в первые дни). Сегодня сеть — это опасное, враждебное место, наполненное спамерами и хакерами. Все чаще она описывается как зона боевых действий, а в зоне военных действий есть потери среди мирных жителей. Трудно писать сетевые сервера безопасно, особенно если протокол хоть чуть-чуть сложен. Практически все программы имели как минимум небольшие проблемы; даже обычные реализации TCP/IP были успешно атакованы. Высокоуровневые языки показали, что также не являются панацеей, и даже добавили свои собственные уязвимости. Появилась необходимая фраза-предупреждение: “Не доверяй всем вводимым данным”, не важно от кого они получены. Недоверие к вводимым данным включает в себя и вторичные данные, например, от DNS-серверов и фильтров milter. Как и большинство первых сетевых программ sendmail был слишком доверчивым в своих первых версиях.

Но самой главной проблемой sendmail было то, что ранние версии работали под администратором. Права администратора были необходимы для открытия слушающего сокета SMTP, для чтения информации о передаче почты индивидуальных пользователей и для доставки почты в почтовые ящики индивидуальных пользователей и их домашние папки. Однако на большинстве сегодняшних систем концепция почтового ящика была отделена от концепции пользователя системы, что эффективно исключило необходимость в административном доступе, за исключением открытия слушающего сокета SMTP. Сегодня sendmail имеет возможность отказываться от прав администратора до того, как начинает обрабатывать соединение, исключая эту проблему для тех систем, которые могут поддерживать такую возможность. Стоит заметить, что на тех системах, которые не доставляют почту прямо в почтовые ящики пользователей, sendmail также может запускаться в окружении с запущенной программой chroot, что дает возможность дальнейшей изоляции прав пользователей.

К сожалению, по мере того, как sendmail получал репутацию программы с низкой безопасностью, его начали обвинять в проблемах, которые к нему не имели отношения. Например, один системный администратор дал запись к своей папке /etc для всего мира и затем обвинил sendmail, когда кто-то заменил ему файл /etc/passwd. Подобные инциденты заставили нас серьезно заняться безопасностью, включая явную проверку владельца и режимов файлов и папок, к которым у sendmail был доступ. Эти опции были настолько драконовскими, что мы были вынуждены добавить параметр DontBlameSendmail для возможности их отключения.

Есть и другие аспекты безопасности, которые не относятся к защите адресного пространства самой программы. Например, увеличение количества спама вызвало увеличение сбора адресов. Команды SMTP VRFY и EXPN были специально спроектированы для валидации индивидуальных почтовых адресов и расширения содержимого почтовых списков соответственно. Эти команды впоследствии стали настолько широко использоваться спамерами, что большинство сайтов их полностью отключило. Это прискорбно, так как данная команда иногда использовалась некоторыми анти-спам агентами для валидации подразумеваемых адресов отправки.

Аналогично, защита от вирусов какое-то время рассматривалась как проблема десктоп-приложений, но сейчас проблема выросла до такой степени, что сейчас любой коммерческий агент передачи почты должен иметь антивирусную проверку. Остальные требования безопасности на сегодняшний день включают обязательное шифрование важных данных, защиту от потери данных и принудительное исполнение законных требований, например, для закона о защите информации о здоровье.

Одним из принципов, которым sendmail следовал с самого начала, была надежность – каждое сообщение должно было быть доставлено либо возвращено обратно отправителю. Но проблема Joe-jobs (когда атакующий подделывает обратный адрес сообщения, рассматриваемая многими как уязвимость) привела к тому, что многие сайты отключили возможность отправки уведомлений об отклонении доставки. Если проблема с доставкой может быть определена пока еще SMTP-соединение открыто, сервер может сообщить о проблеме неисполнением команды, но после того, как SMTP-соединение закрыто, неверно адресованное сообщение тихо исчезнет.

На сегодняшний день большинство легальных писем доставляются за один раз, поэтому информация о проблемах отправки получается, однако в принципе в мире главной стала позиция «безопасность важнее надежности».

17.7. Эволюция sendmail

Программное обеспечение не выживает в быстро изменяющемся окружении без изменений для приспособления к этому окружению. Появляются новые технологии, которые оказывают влияние на операционные системы, которые в свою очередь влияют на библиотеки и фреймворки, а те – на приложения. Если приложение имеет успех, его начинают использовать в еще более проблемном окружении. Изменения неизбежны; для того, чтобы преуспеть, вам необходимо принять и использовать это. Эта секция описывает наиболее важные изменения в sendmail, которые происходили по мере его развития.

17.7.1. Конфигурирование становится более подробным

Изначально конфигурация sendmail была довольно лаконичной. Например, имена опций и макросы все состояли из одного символа. На это было три причины. Первая, это делало парсинг очень простым (важно в 16-битном окружении). Второе, число опций было не очень большим, поэтому было не сложно придумать мнемонические имена. В-третьих, уже существовал порядок использования одиночных символов как флагов командной строки.

Аналогично, наборы правил преобразования изначально были под номерами, а не под названиями. Это, наверное, было допустимо с небольшим набором правил, но с ростом их числа, стало важно, чтобы у них были мнемонические имена.

По мере того, как окружение в котором работал sendmail становилось более сложным и 16-битное окружение исчезало, необходимость более богатой конфигурации стала очевидной. К счастью, было возможно добавлять изменения с сохранением обратной совместимости. Эти изменения кардинально улучшили читаемость конфигурационного файла.

17.7.2. Большее количество соединений с другими подсистемами: большая интеграция

Когда sendmail был написан, почтовая система была главным образом изолирована от остальной операционной системы. Существовало несколько служб, которые требовали интеграции, например, файлы /etc/passwd и /etc/hosts. Переключатели служб не были еще изобретены, службы каталогов не существовали, конфигурационные файлы были небольшими и поддерживались вручную.

Но все быстро изменилось. Одним из первых нововведений были DNS. Хотя абстракция поиска системного хоста (gethostbyname) работала для поиска IP-адресов, электронная почта должна была использовать другие запросы, такие как MX. Позже IDA sendmail включила функциональность поиска по внешним базам данных, используя файлы dbm(3). Sendmail 8 получил в обновлении общую службу сопоставления данных, которая позволяла взаимодействие с другими типами баз данных и внутренними данными, с которыми нельзя было работать, используя простые преобразования (например, деэкранирование в адресах).

Сегодня почтовая система зависит от многих внешних служб, которые в общем не созданы специально для эксклюзивного использования электронной почтой. Это привело к тому, что в коде sendmail появилось больше абстракций. Это также привело к тому, что поддержка кода почтовой системы стала сложнее, так как количество изменяющихся элементов прибавилось.

17.7.3. Адаптация к враждебному миру

Sendmail был разработан в мире, который кажется абсолютно чужим по современным стандартам. Пользовательский контингент в ранние периоды жизни сетей состоял главным образом из исследователей, которые были относительно добродушными, несмотря на подчас яростные академические споры. Sendmail отражал тот мир, в котором он был создан, большой акцент был сделан на надежную доставку почты, даже при наличии пользовательских ошибок.

Сегодняшний мир намного более враждебен. Большая доля электронной почты идет от злоумышленников. Задачи агента передачи почты сместились от получения почты до защиты от плохой почты. Фильтрация стала, пожалуй, одной из главных задач для агента передачи на сегодняшний день. Это потребовало множества изменений в sendmail.

Например, были добавлены многие наборы правил, чтобы разрешить проверку параметров входящих команд SMTP, это дало возможность отлавливать проблему как можно раньше. Гораздо проще отклонить сообщение на стадии чтения конверта, чем после того, как вы направили ресурсы на чтение всего сообщения, и еще более сложно это сделать, когда вы приняли сообщение для доставки. Изначально фильтрация в основном делалась путем принятия сообщения, передачи его фильтрующей программе и затем отправки сообщения другому экземпляру sendmail, если сообщение проходило фильтр (так называемая “сэндвич”-конфигурация). На сегодняшний день это слишком неэффективно.

Аналогично, sendmail прошел путь от довольно простого потребителя TCP/IP соединений до гораздо более сложной системы, способной “просматривать” данные, получаемые из сети, чтобы понять, передает ли отправитель команды до того, как предыдущая команда была принята. Это разрушает некоторые предыдущие абстракции, которые были сделаны для того, чтобы sendmail работал в различных типах сетей. Сегодня потребовалось бы немало работы для адаптации sendmail к сети XNS или DECnet, так как информация о работе с TCP/IP уже встроена в большую часть кода.

Многие функции конфигурирования были добавлены для защиты от враждебного мира, среди них таблицы доступа, список черных дыр в реальном времени, подавление сбора адресов, защита от отказа в обслуживании и фильтрация спама. Это значительно усложнило задачу конфигурирования почтовых систем, но стало абсолютно необходимым для адаптации к сегодняшнему миру.

17.7.4. Внедрение новых технологий

Многие новые стандарты появились за эти годы, они потребовали серьезных изменений в sendmail. Например, добавление TLS (шифрования) потребовало значительных изменений в большей части кода. Конвейерная обработка SMTP (pipelining) потребовала низкоуровневого взаимодействия с потоком TCP/IP для избежания дедлоков. Добавление порта отправки (587) требовало способности слушать несколько входящих портов, включая поддержку различного поведения, в зависимости от входящего порта.

Некоторые ограничения были вызваны скорее обстоятельствами, чем стандартами. Например, добавление интерфейса фильтров milter было прямым ответом на спам. Хотя milter не был опубликованным стандартом, это была новая большая технология.

Во всех случаях эти изменения определенным образом улучшали почтовую систему: либо с точки зрения безопасности, либо за счет производительности или путем добавления новой функциональности. Однако, все изменения имели свою цену, усложняя код и конфигурационный файл.

17.8. Что если бы я делал sendmail сегодня?

Многие вещи на сегодняшний день я бы сделал по-другому. Некоторые было трудно предсказать в то время (например, как спам изменит восприятие почтовой системы, как будут выглядеть современные наборы инструментов и т.д.), а некоторые были очень предсказуемы. В процессе написания sendmail я многое изучил о e-mail, TCP/IP и программировании вообще. Все растут в процессе того, как пишут код.

Но есть множество вещей, которые я бы сделал также, некоторые из них даже несмотря на противоречие с общепринятым мнением.

17.8.1. Что бы я сделал иначе

Наверное моей главной ошибкой с sendmail было то, что я не смог распознать достаточно быстро, насколько важной станет программа. У меня было несколько возможностей подтолкнуть мир в верном направлении, но я ими не воспользовался; в некоторых случаях даже наоборот, я нанес вред, например, не сделав sendmail более строгим к неверным данным, когда это следовало сделать. Я также довольно рано понял, что синтаксис конфигурационного файла нужно улучшить, когда работало только несколько сотен экземпляров sendmail, но решил ничего не менять, потому что не хотел причинять неудобства пользователям. Позже стало ясно, что лучше было бы исправить все раньше, пусть ценой небольшого неудобства, но зато сделать лучше в долгосрочной перспективе.

Синтаксис почтовых ящиков версии 7

Одним из примеров было то, как в почтовых ящиках версии 7 разделялись сообщения. Они использовали строку, начинавшуюся с “From?” (где “?” это ASCII символ пробела, 0x20) для разделения сообщений. Если приходило сообщение, содержащее слово “From?” в начале строки, программа локальной почты конвертировала ее в “>From?”. После усовершенствования на некоторых системах стала требоваться предварительно пустая строка, но на это нельзя было рассчитывать. И по сей день “>From” появляется в чрезвычайно непредсказуемых местах, которые не совсем связаны с email (но очевидно были обработаны email в какой-то момент времени). В ретроспективе я понимаю, что, наверное мог бы исправить почтовую службу BSD — сделать использование нового синтаксиса. В то время меня бы все проклиниали, но в перспективе я спас бы мир от огромных проблем.

Синтаксис и содержимое конфигурационного файла

Возможно, моей самой большой ошибкой в синтаксисе конфигурационного файла было использование таба (HT, 0x09) в правилах преобразования для разделения паттерна от заменяющего текста. В то время я делал подобно тому, как сделано в make, и только годы спустя я узнал, что Стюарт Фельдман, автор make, считал это одной из своих самых больших ошибок. Помимо того, что когда вы смотрите на экран, совсем неочевидно, если именно символ табуляции используется в файле, эти символы не сохраняются после вырезания и вставки на большинстве Windows-систем.

Хотя я считаю, что использование правил преобразования было хорошей идеей (см. ниже), я бы изменил общую структуру конфигурационного файла. Например, я не ожидал необходимость иерархии в конфигурации (например, опций, которые будут задаваться по-разному для различных портов SMTP). В то время, когда создавался конфигурационный файл, не было “стандартных”

форматов. Сегодня бы склонялся к тому, чтобы сделать конфигурацию подобно Apache – она чистая, аккуратная и достаточно выразительная – или даже может быть включил язык, подобный Lua.

Когда разрабатывался sendmail, пространство адресов было небольшим и протоколы были еще меняющимися. Большие затраты сил на конфигурационный файл казались хорошей идеей. Сегодня, это выглядит как ошибка: у нас огромное пространство адресов (для агентов передачи почты), а стандарты довольно статичные. Более того, часть “конфигурационного файла” — на самом деле код, который нужно обновлять в новых релизах. Конфигурационный файл .mc исправляет это, но необходимость пересобирать конфигурацию каждый раз, когда обновляется программа, это неудобство. Простым решением для этого было бы иметь два конфигурационных файла, читаемые sendmail’ом: один скрытый и устанавливаемый с каждым новым релизом, а другой открытый, используемый для локальной конфигурации.

Использование инструментов

Сегодня доступно множество новых инструментов, например, для конфигурирования и собирания программ. Инструменты могут помочь в этом, но они также могут быть излишними, затрудняя понимание системы. Например, никогда не нужно использовать грамматику yacc(1), если все что вам нужно — это strtok(3). Но и переизобретать колесо также не является хорошей затеей. В частности, несмотря на некоторые замечания, я все равно точно бы использовал сегодня autoconf.

Обратная совместимость

Если бы у меня была возможность заглянуть в будущее и узнать, насколько вездесущим станет sendmail, я бы не беспокоился о том, что придется прекратить поддержку существующих инсталляций в первые дни разработки. Если существующая практика имеет серьезные проблемы, ее нужно исправлять, а не подстраиваться под нее. Несмотря на это, я бы все равно не сделал бы строгую проверку всех форматов сообщений; некоторые проблемы могут быть легко и безопасно проигнорированы или пропатчены. Например, я бы все равно добавил поле заголовка Message-Id: в сообщения, у которых его не было, но я бы больше склонялся к тому, чтобы отклонять сообщения без поля заголовка From:, а не пытался бы создавать его из информации в конверте.

Внутренние абстракции

Есть несколько внутренних абстракций, к которым я бы не прибегал сегодня, а есть другие, которые бы добавил. Например, я бы не использовал строки с завершающим нулем, вместо этого предпочел бы пары “длина/значение”, несмотря на то, что это означало бы, что большую часть стандартной библиотеки C было бы трудно использовать. Но даже только последствия этого решения для безопасности стоили того. Однако, я бы не пытался создавать систему обработки ошибок на C, вместо этого создал бы последовательную систему статусных кодов, которые бы использовались в коде программы вместо создания процедур, возвращающих null, false или отрицательные числа для представления ошибок.

Я бы точно абстрагировал бы концепцию имен почтовых ящиков от ID пользователя в Unix. В то время, когда я писал sendmail, модель была такова, что сообщения отправлялись только пользователям Unix. Сегодня такое не происходит; даже в системах, которые используют эту модель, существуют системные аккаунты, которые никогда не должны получать e-mail.

17.8.2. Что я бы оставил точно так же

Конечно, были и удачные решения...

Syslog

Одним из успешных сайд-проектов sendmail был syslog. В то время, когда писался sendmail, программы, которым нужно было писать информацию в лог, имели для этого специальный файл. Такие файлы были разбросаны по системе. Syslog было трудно написать в то время (UDP еще не существовало, поэтому я использовал так называемые трх файлы), но это того стоило. Однако, я бы добавил одно существенное изменение: я бы обратил больше внимания на то, чтобы сделать синтаксис логируемых сообщений пригодным для разбора – я все-таки не смог предсказать существование мониторинга логов.

Правила преобразования

Правила преобразования были во многом опорочены, но я бы снова использовал их (хотя, возможно, не во всех случаях, где они используются сейчас). Использование символа табуляции было явной ошибкой, но учитывая ограничения ASCII и синтаксиса почтовых адресов, какой-то символ экранирования все-таки необходим. В общем, концепция использования замены по совпадению с паттерном работала хорошо и была очень гибкой.

Избегать ненужных инструментов

Несмотря на мой комментарий выше о том, что я бы использовал больше внешних инструментов, я бы не стал использовать многие доступные сейчас динамические библиотеки. На мой взгляд многие из них слишком раздуты. Библиотеки нужно выбирать осторожно, балансируя между пользой от повторного использования кода и проблемой использования чересчур мощного инструмента для решения простой проблемы. Как минимум, я бы избегал использования XML в качестве языка конфигурации. Я думаю, что его синтаксис слишком расточителен для таких задач. У XML есть свое место, но сейчас он используется чересчур часто.

Код на С

Некоторые люди предложили, что более естественным языком для реализации были бы Java или C++. Несмотря на хорошо известные проблемы C, я все равно выбрал бы его для реализации. Отчасти это навеяно личными предпочтениями: я знаю С намного лучше, чем Java или C++. Но я также разочарован той небрежностью, с которой большинство объектно-ориентированных языков относится к распределению памяти. Распределение памяти во многом затрагивает проблемы производительности, которые трудно охарактеризовать. Sendmail используется объектно-ориентированные концепции внутри, где это необходимо (например, реализация классов преобразования данных), но на мой взгляд полный переход на объектно-ориентированное программирование было бы очень расточительным и чересчур ограничивающим.

17.9. Заключение

Агент передачи почтовых сообщений sendmail появился в момент невероятного подъема, своего рода “на диком Западе”, существовавшем, когда e-mail не был должным образом продуман, а текущие почтовые стандарты не были еще сформулированы. В последующие 31 год “почтовая проблема” изменялась от просто надежной работы до работы с большими сообщениями и серьезными нагрузками, затем до защиты узлов от спама и вирусов и, наконец, на данный момент – использования в качестве платформы для огромного количества приложений, основанных на использовании e-mail. Sendmail эволюционировал в рабочую лошадку, используемую даже самыми избегающими рисков корпорациями, после того, как электронная почта прошла путь от простой текстовой коммуникации между людьми до части инфраструктуры, использующей мультимедиа и критичной для выполнения задач.

Причины такого успеха не всегда очевидны. Создание небольшой группой частично занятых в проекте разработчиков программы, которая выживает и даже процветает в постоянно меняющемся

мире, не может быть выполнено при использовании обычных методологий разработки программного обеспечения. Я надеюсь, что пролил некоторый свет на факторы успеха sendmail.

Ссылки

- <http://ftp.isc.org/www/survey/reports/2011/01/>
- <http://www.sendmail.com/>
- http://doc.cat-v.org/bell_labs/upas_mail_system/upas.pdf
- <http://milter.org/>
- <http://postfix.org/>

18. SnowFlock

Облачные технологии позволяют создавать привлекательные и экономичные вычислительные платформы. Вместо покупки и настройки физического сервера с соответствующими затратами времени, усилий и финансов, пользователи могут арендовать "серверы" в облаке с помощью нескольких кликов мышью за сумму менее чем 10 центов в час. Провайдеры облачных технологий поддерживают низкий уровень цен, предоставляя в распоряжение пользователей виртуальные машины (virtual machines - VM) вместо физических систем. Ключевым компонентом является программное обеспечение для виртуализации, называемое монитором виртуальной машины (virtual machine monitor - VMM) и позволяющее эмулировать физическую машину. Пользователи изолированы безопасным образом в рамках своих "гостевых" виртуальных машин и не беспокоятся о том, что они делят используемую физическую машину ("узел") с множеством других пользователей.

18.1. Знакомство с SnowFlock

Использование облачных технологий позволяет повысить скорость работы организации. В случае использования физических серверов пользователям приходилось с нетерпением ждать момента, когда представители организации (не торопясь) подтверждают покупку сервера, разместят заказ, доставят сервер и установят операционную систему вместе с наборами приложений. Вместо ожидания завершения работы представителей организации, затягивающегося на недели, пользователь облака самостоятельно контролирует процесс его работы и может создать новый отдельный сервер за считанные минуты.

К сожалению, группы облачных серверов являются разделенными. Используя технологии быстрой установки и модель оплаты за используемые ресурсы, облачные серверы обычно являются элементами групп различного количества аналогично настроенных серверов, выполняющих динамические и масштабируемые задачи, связанные с параллельными вычислениями, обработкой данных или обслуживанием веб-страниц. Так как серверы в этих группах периодически используют для загрузки один и тот же постоянный образ, коммерческие облачные системы не предоставляют полноценного решения для проведения вычислений в зависимости от потребностей в мощностях. После создания сервера пользователь облака должен также установить его принадлежность к кластеру и провести дополнительные работы, связанные с добавлением новых серверов.

SnowFlock решает эти проблемы с помощью технологии клонирования виртуальных машин (VM Cloning), представленной вызовом API. Аналогично тому, как код приложения вызывает службы ОС с помощью интерфейса системных вызовов при повседневной работе, на сегодняшний день он также может вызывать службы облачных сервисов, используя подобный интерфейс. С помощью технологии клонирования виртуальных машин операции резервирования ресурсов, управления кластером, а также логика приложения могут быть объединены программно и рассматриваться как отдельная логическая операция.

Использование вызова для клонирования виртуальных машин позволяет получить идентичные на момент клонирования копии виртуальной машины родительского сервера на множестве облачных серверов. Логически клонированные виртуальные машины наследуют данные состояния родительской виртуальной машины, включая кэши уровня операционной системы и приложений. Более того, клонированные виртуальные машины автоматически добавляются во внутреннюю локальную сеть, таким образом входя в состав динамически масштабируемого кластера. Новые вычислительные ресурсы, представленные идентичными виртуальными машинами, могут быть созданы мгновенно, а также могут динамически загружаться работой в случае необходимости.

На практике метод клонирования виртуальных машин зарекомендовал себя как успешно функционирующая, эффективная и быстрая технология. В данной главе мы опишем метод эффективной интеграции реализации технологии клонирования виртуальных машин SnowFlock в различные программные модели и фреймворки, метод реализации данной технологии с учетом минимизации затрат ресурсов программным окружением и системой, а также метод создания множества новых виртуальных машин в течение пяти секунд или меньшего промежутка времени.

Учитывая наличие API для программного контроля за клонированием виртуальных машин и наличие биндингов для языков C, C++, Python и Java, SnowFlock является весьма гибким и многоцелевым программным компонентом. Мы успешно использовали SnowFlock в реализации прототипов нескольких значительно отличающихся друг от друга систем. В условиях параллельных вычислений мы достигли превосходных результатов, клонируя работающие виртуальные машины, которые совместно распределяли нагрузку между множеством физических узлов. При работе с приложениями для параллельных вычислений мы использовали интерфейс передачи сообщений (Message Passing Interface - MPI) и запускали их на кластере выделенных серверов, при этом модифицировав метод запуска интерфейса передачи сообщений для того, чтобы повысить производительность немодифицированных приложений и снизить затраты ресурсов, предоставляя кластер из только что клонированных виртуальных по запросу при каждом запуске. Наконец, в значительном отличии от других случаев, мы использовали SnowFlock для повышения эффективности и производительности elastic-серверов. На сегодняшний день облачные elastic-серверы осуществляют холодную загрузку новых рабочих серверов при необходимости обработки повышенных нагрузок. Применяя вместо этого технологию клонирования виртуальных машин, SnowFlock позволяет вводить в строй новые рабочие серверы в 20 раз быстрее, а так как клонированные виртуальные машины наследуют рабочие буферы от родительской виртуальной машины, они быстрее достигают своей пиковой производительности.

18.2. Технология клонирования виртуальных машин

Как становится ясно из названия, клонированные виртуальные машины (практически) полностью идентичны родительской виртуальной машине. Существуют некоторые незначительные, но необходимые различия для решения таких проблем, как конфликты MAC-адресов, но мы вернемся к рассмотрению этого вопроса позднее. Для создания клонированной виртуальной машины данные локального диска и оперативной памяти должны быть доступны в полном объеме, что заставляет нас рассмотреть первое важное архитектурное решение: должны ли мы скопировать эти данные немедленно или по мере необходимости?

Простейшим способом реализации механизма клонирования виртуальной машины является реализации стандартной возможности "миграции" виртуальной машины. Обычно миграция осуществляется в том случае, когда работающая виртуальная машина должна быть перенесена на другой узел, например, в том случае, когда узел находится в перегруженном состоянии или может быть отключен для технического обслуживания. Так как виртуальная машина является программным решением, ее состояние может быть сохранено в файле данных, после чего этот файл может быть скопирован на новый, более подходящий узел, на котором работа виртуальной машины продолжится после кратковременного перерыва. Для выполнения этой задачи существующие мониторы виртуальных машин создают файл, содержащий "данные контрольной точки" виртуальной машины,

включающие в себя локальную файловую систему, образ памяти, регистры виртуального центрального процессора (VCPU), и.т.д. При миграции новая загруженная копия подменяет оригинал, но процесс может быть модифицирован таким образом, чтобы система клонировалась, а работа оригинальной системы не прерывалась. В ходе этого "активного" процесса все данные виртуальной машины передаются незамедлительно, что позволяет достичь лучшей начальной производительности. Недостатком активной репликации является тот факт, что сложный процесс копирования всех данных виртуальной машины должен быть завершен перед началом работы виртуальной машины, что значительно замедляет инициализацию.

Диаметрально противоположным способом, реализованным в SnowFlock, является "отложенная" репликация данных. Вместо копирования всех данных, которые могут понадобиться виртуальной машине, SnowFlock передает только необходимые для запуска данные, а остальные данные передаются позднее, когда они понадобятся клонированной виртуальной машине. Этот способ имеет два преимущества. Во-первых, он позволяет минимизировать задержку, выполняя незамедлительно настолько малое количество работы, насколько это возможно. Во-вторых, он позволяет повысить эффективность работы, копируя только те данные, которые действительно используются клонированной виртуальной машиной. Повышение эффективности работы клонированной виртуальной машины, конечно же, зависит от выполняемой задачи, но только малая часть приложений использует каждую страницу памяти и каждый файл из локальной файловой системы.

Однако, достоинства отложенной репликации не обуславливают отсутствие недостатков. Так как передача данных откладывается до последнего, клонированная виртуальная машина будет ожидать их приема до момента продолжения работы. Эта ситуация аналогична сохранению страниц памяти в раздел подкачки диска в многозадачной системе: приложения блокируются, ожидая получения данных из источника с длительной задержкой. В случае SnowFlock блокировка в некоторой степени снижает производительность клонированной виртуальной машины; степень снижения производительности зависит от приложения. Для высокопроизводительных вычислительных приложений мы не обнаружили значительного снижения производительности, но производительность клонированного сервера базы данных в первое время будет низкой. Следует учесть, что этот эффект кратковременен: он длится несколько минут, после чего все необходимые данные передаются и производительность клонированной виртуальной машины сравнивается с производительностью родительской виртуальной машины.

Кстати, если вы располагаете большим опытом работы с виртуальными машинами, вас наверняка интересует вопрос о том, будут ли полезны оптимизации, используемые при миграции "в реальном времени" в данном случае. Процесс миграции в реальном времени оптимизируется с целью сокращения интервала между отключением оригинальной виртуальной машины и возобновлением выполнения работы с помощью ее копии. Для этого монитор виртуальной машины (VMM) предварительно копирует данные виртуальной машины в то время, как оригинальная машина все еще работает, поэтому после завершения ее работы необходимо передать только недавно измененные страницы памяти. Эта техника не влияет на интервал между запросом миграции и началом выполнения работы копией виртуальной машины, а значит, не способна уменьшить задержку запуска виртуальной машины при "активном" клонировании.

18.3. Подход SnowFlock

SnowFlock реализует операцию клонирования виртуальных машин с помощью примитива под названием "VM Fork", который аналогичен стандартной функции `fork` из состава Unix, но имеет несколько важных отличий. Во-первых, вместо дублирования единственного процесса, VM Fork дублирует всю виртуальную машину, включая всю память, все процессы и виртуальные устройства, а также локальную файловую систему. Во-вторых, вместо создания одной копии процесса, выполняющегося на том же физическом узле, VM Fork может запустить одномоментно множество копий виртуальных машин в параллельном режиме. Наконец, виртуальные машины могут быть

запущены на отдельных физических серверах, позволяя вам быстро расширить возможности обла-ка в случае необходимости.

Данные концепции являются ключевыми для SnowFlock:

- Виртуализация: Виртуальная машина является вычислительным окружением, что делает возможным клонирование обла-ков и машин.
- Отложенное распространение: Данные состояния виртуальной машины не копируются до тех пор, пока они не потребу-ются, поэтому клонированные виртуальные машины становятся работоспособными в течение нескольких секунд.
- Многоадресная передача: Клонированные виртуальные машины испытывают потребность в одних и тех же данных со-стояния. При использовании многоадресной передачи множество клонированных виртуальных машин начнут работать так же быстро, как если бы клонированная виртуальная машина была одна.
- Ошибки страниц: В момент, когда клонированная виртуальная машина пытается получить доступ к отсутствующей стра-нице памяти, возникает ошибка страницы, инициирующая выполнение запроса к родительской виртуальной машине. Процесс работы клонированной виртуальной машины прерывается до того момента, пока необходимая страница не будет получена.
- Копирование при записи (Copy on Write - CoW): Копируя страницы памяти и диска перед их перезаписью, родительская виртуальная машина может продолжать работу, сохраняя данные состояния для использования клонированными виртуальными машинами.

Мы реализовали SnowFlock, используя систему виртуализации Xen, поэтому для лучшего пони-мания полезно привести специфическую для Xen терминологию. В окружении Xen мониторы вир-туальных машин называются гипервизорами, а сами виртуальные машины - доменами. На каждой физической машине (узле) существует привилегированный домен, называемый "domain 0" (dom0) и имеющий полный доступ к узлу и его физическим устройствам, который может использоваться для контроля дополнительных гостевых или "пользовательских" виртуальных машин, называемых "domain U" (domU).

В общих чертах, SnowFlock содержит набор модификаций для гипервизора Xen, позволяющих ему благополучно восстанавливать работу в случае доступа к отсутствующим ресурсам, набор под-держивающих работу процессов и систем, которые выполняются в рамках домена dom0 и совме-стно передают недостающие данные состояния виртуальным машинам, а также некоторые допол-нительные модификации для операционных систем, выполняющихся в клонированных виртуаль-ных машинах. Существует шесть основных компонентов SnowFlock.

- Дескриптор виртуальной машины: Этот небольшой объект используется для запуска клонированной виртуальной маши-ны и содержит ее скелет, необходимый для начала работы. Он не содержит внутренние органы и мышцы, необходимые для выполнения полезной работы.
- Система многоадресного распространения данных (mcdist): Эта система на стороне родительской виртуальной маши-ны эффективно распространяет информацию о состоянии виртуальной машины одновременно между всеми клонирован-ными виртуальными машинами.
- Процесс сервера памяти: Этот процесс на стороне родительской виртуальной машины поддерживает постоянную копию данных состояния виртуальной машины и делает ее доступной для всех клонированных виртуальных машин при необхо-димости с помощью mcdist.
- Процесс memtap: этот процесс на стороне клонированной виртуальной машины обслуживает данную виртуальную ма-шину и связывается сервером памяти для запроса страниц, которые необходимы, но отсутствуют.
- Оптимизация клонированных виртуальных машин: Гостевое ядро ОС, работающее в клонированных виртуальных маши-нах, может способствовать уменьшению интенсивности передач данных состояния виртуальных машин по запросам, от-правляя подсказки монитору виртуальной машины. Это необязательная функция, но ее использование весьма желательно для повышения эффективности работы.
- Стек управления: Демоны, выполняющиеся на каждом физическом узле и предназначенные для распределения задач ме-жду другими компонентами и управления родительскими и клонированными виртуальными машинами SnowFlock.

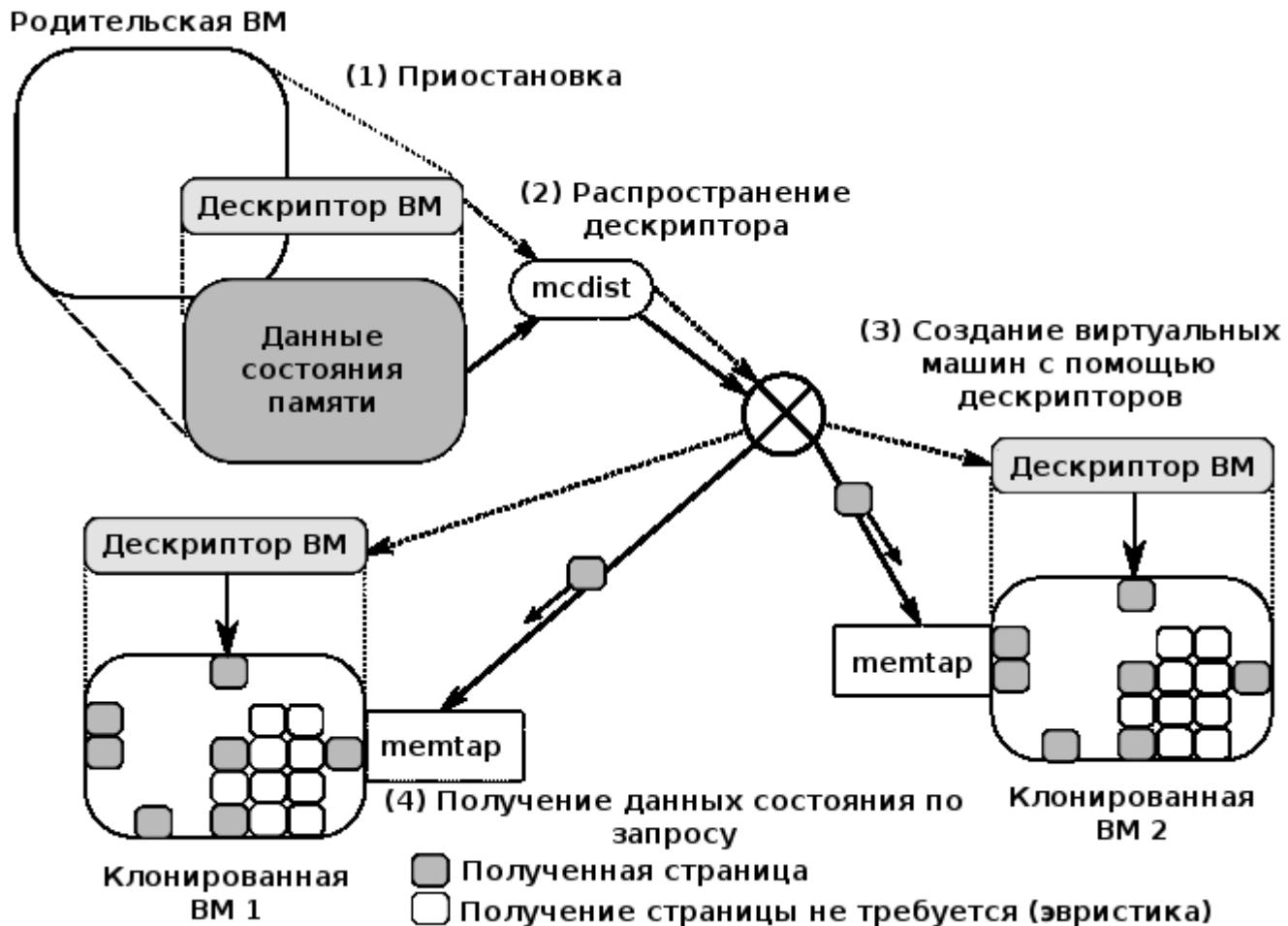


Рисунок 18.1: Архитектура системы репликации виртуальных машин в SnowFlock

Образно говоря, [Рисунок 18.1](#) отражает процесс клонирования виртуальной машины, обозначая четыре основных шага: (1) остановка родительской виртуальной машины для создания архитектурного дескриптора; (2) распространение этого дескриптора среди всех целевых узлов; (3) создание клонированных виртуальных машин, практически не располагающих данными состояния, а также (4) распространение данных состояния по запросам. Данный рисунок также отражает использование системы многоадресного распространения данных `mcdist` и предупреждение запросов с помощью механизмов оптимизации на стороне гостевых виртуальных машин.

Если вы желаете испытать SnowFlock в рабочих условиях, вы можете получить данный программный продукт двумя способами. Документация и открытый исходный код из состава оригинального исследовательского проекта SnowFlock Университета Торонто находятся в свободном доступе¹. Если вы предпочитаете версию, используемую в индустриальных проектах, бесплатные лицензии для некоммерческого использования распространяются компанией GridCentric Inc.² Так как SnowFlock содержит модификации для гипервизора и требует доступа к домену `dom0`, для установки SnowFlock требуется привилегированный доступ к машинам. По этой причине вам придется использовать собственное аппаратное обеспечение и вы не сможете испытать данный продукт, являясь пользователем такого коммерческого облачного окружения, как Amazon EC2.

В нескольких следующих разделах мы опишем различные программные компоненты, взаимодействующие с целью реализации быстрого и эффективного процесса клонирования виртуальных машин. Все компоненты, которые мы опишем, взаимодействуют так, как показано на [Рисунке 18.2](#).

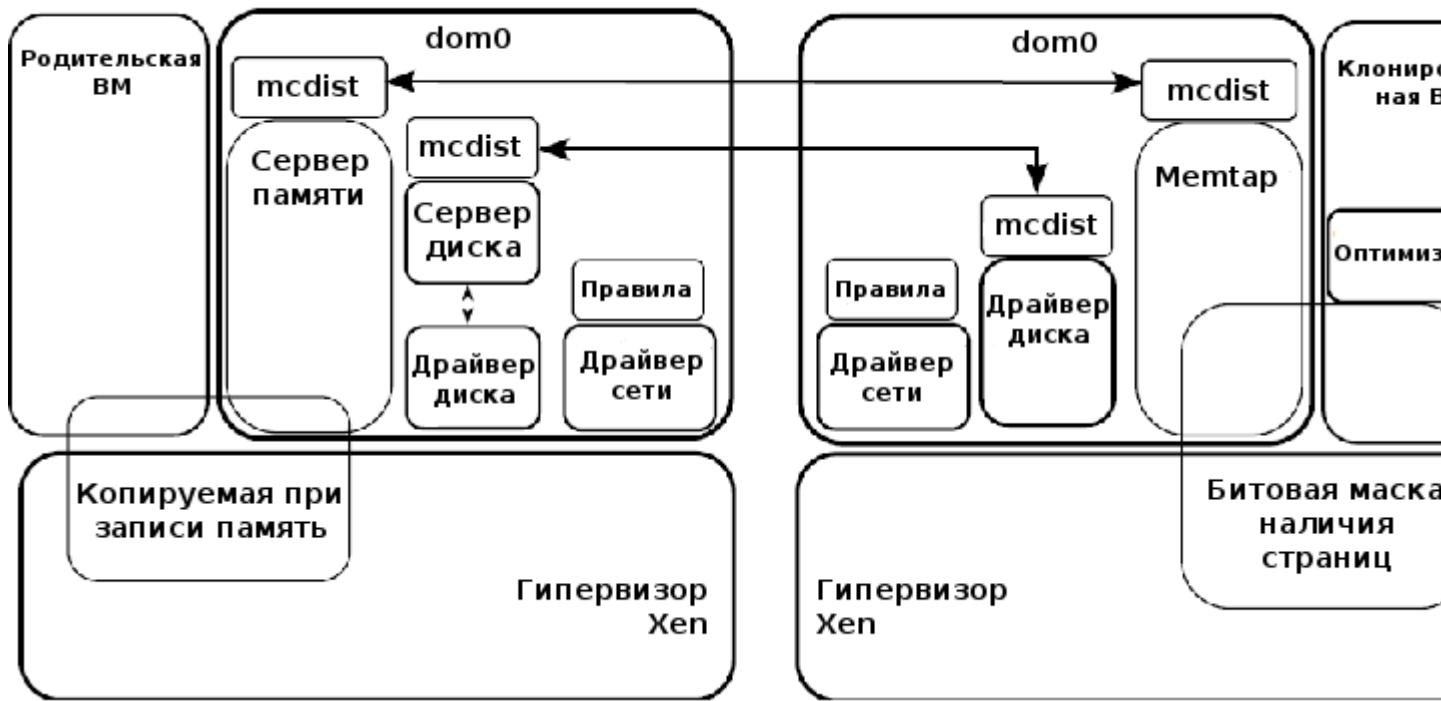


Рисунок 18.2: Программные компоненты из состава SnowFlock

18.4. Архитектурный дескриптор виртуальной машины

Ключевым архитектурным решением, примененным в SnowFlock, является возможность выполнения отложенной репликации данных состояния виртуальной машины во время ее работы. Другими словами, копирование памяти виртуальной машины является операцией позднего связывания, предоставляющей множество возможностей для оптимизации.

Первым шагом для проведения в жизнь этого архитектурного решения является генерация архитектурного дескриптора для данных состояния виртуальной машины. Это данные, которые будут использованы для создания клонированных виртуальных машин. Дескриптор содержит абсолютный минимум необходимых для создания виртуальной машины и планирования ее работы данных. Как становится ясно из названия, этот абсолютный минимум данных состоит из структур данных, необходимых для удовлетворения требований используемой архитектуры. В случае SnowFlock под архитектурными требованиями понимают комбинацию требований процессора x86 и требований технологии Xen. Таким образом, архитектурный дескриптор содержит такие структуры данных, как таблицы страниц, виртуальные регистры, метаданные устройств, метки времени, и.т.д. Интересующемуся читателю следует обратиться к [LCWB+11] для ознакомления с более подробным описанием содержимого архитектурного дескриптора.

Архитектурный дескриптор имеет три важных свойства: Во-первых, он может быть создан за короткий промежуток времени; нередко это промежуток времени длительностью 200 миллисекунд. Во-вторых он имеет малый размер, обычно на три порядка меньше, чем объем памяти, зарезервированной во время работы родительской виртуальной машины (1 МБ для 1 ГБ памяти виртуальной машины). И в-третьих, клонированная виртуальная машина может быть создана из дескриптора менее чем за секунду (обычно за 800 миллисекунд).

Последствием, конечно же, является тот факт, что клонированные виртуальные машины не обладают большинством своих данных состояния в момент их создания из дескриптора. В следующих разделах мы опишем способ решения этой проблемы, а также способ реализации возможностей оптимизации, предоставляемых данным подходом.

18.5. Компоненты на стороне родительской виртуальной машины

Как только происходит клонирование виртуальной машины, она становится родительской для дочерних или клонированных виртуальных машин. Как и все ответственные родители, она должна следить за благополучием своих потомков. Она делает это с помощью ряда служб, посредством которых производится доставка данных состояния памяти и диска клонированным виртуальным машинам по запросу.

18.5.1. Процесс сервера памяти

Когда архитектурный дескриптор создан, виртуальная машина прерывает свою работу. Таким образом фиксируются данные состояния памяти; перед приостановкой работы виртуальной машины и планированием этого действия, внутренние драйверы операционной системы сохраняют ее состояние, из которого клонированные виртуальные машины смогут соединиться с внешним миром, работая в новом окружении. Мы использовали это состояние для создания "сервера памяти", или службы `memserver`.

Сервер памяти будет предоставлять всем клонированным виртуальным машинам участки памяти, которые они потребуют от родительской виртуальной машины. Единица распространения данных памяти идентична странице памяти архитектуры x86 (4 килобайта). В простейшем случае сервер памяти ожидает запросов страниц от клонированных виртуальных машин и обслуживает одну страницу памяти и одну виртуальную машину в каждый момент работы.

Однако, эта память используется родительской виртуальной машиной, которая должна продолжить свою работу. Если мы позволим родительской виртуальной машине просто модифицировать эту память, мы будем предоставлять поврежденные участки памяти клонированным виртуальным машинам: переданный участок будет отличаться от участка во время клонирования, поэтому процесс работы клонированных виртуальных машин будет серьезно нарушен. Используя терминологию разработчиков ядра, можно сказать, что это верный способ для создания трассировок стека.

Для преодоления данной проблемы может использоваться классический для операционных систем подход: метод копирования при записи или копируемая при записи память. При поддержке гипервизора Xen мы можем убрать привилегии записи со всех страниц памяти родительской виртуальной машины. Когда родительская виртуальная машина попытается модифицировать одну из страниц памяти, будет сгенерирована аппаратная ошибка доступа к ней. Система Xen располагает информацией о причинах данной ошибки и создаст копию страницы. Родительская виртуальная машина получит разрешение на перезапись данных оригинальной страницы и продолжит выполнение в то время, как сервер памяти будет использовать копию страницы, доступную только для чтения. Таким образом, состояние памяти с момента клонирования остается неизменным, работа клонированных виртуальных машин не нарушается и родительская виртуальная машина может продолжать свою работу. Дополнительные затраты ресурсов за счет использования копирования при записи минимальны: аналогичные механизмы используются ядром Linux, например, при создании новых процессов.

18.5.2. Многоадресное распространение данных с помощью службы `mcdist`

Клонированные виртуальные машины обычно страдают эзистенциальным синдромом, известным под названием "детерминизм судьбы". Мы создаем клонированные виртуальные машины с одной и той же целью: например, для выравнивания цепочек X-хромосом ДНК относительно сегмента цепочки Y-хромосом из базы данных. Более того, мы создаем множества клонированных виртуальных машин для выполнения одинаковых действий, возможно, выравнивания одних и тех же цепочек X-хромосом относительно различных сегментов цепочек из базы данных или выравнивания различных цепочек относительно одного и того же сегмента цепочки Y-хромосом. В данном

случае большая часть попыток доступа к памяти клонированными виртуальными машинами будет идентично локализована: они исполняют одинаковый код и используют большие объемы сходных данных.

Для эксплуатации идентичной локализации требуемых страниц памяти в состав SnowFlock введена наша реализация многоадресной системы распространения данных `mcdist`. Служба `mcdist` использует многоадресную передачу по протоколу IP для одновременной доставки пакетов данных множеству принимающих сторон. При ее функционировании используются преимущества параллельной работы аппаратной реализации сетевого интерфейса с целью снижения нагрузки на сервер памяти. После отправки ответа на первый запрос страницы памяти всем клонированным виртуальным машинам, каждый запрос от клонированной виртуальной машины позволяет сохранить страницу другим клонированным виртуальным машинам из-за их идентичных методов работы с памятью.

В отличие от других систем многоадресной передачи, `mcdist` не является надежной, не доставляет пакеты в заданной последовательности и не позволяет атомарно доставить ответ всем принимающим сторонам. Многоадресная передача введена исключительно как мера оптимизации и надежная доставка страницы должна быть осуществлена исключительно той клонированной виртуальной машине, которая явно ее запросила. Таким образом, архитектура становится элегантной и простой: сервер просто осуществляет многоадресную передачу ответов, а клиенты устанавливают период времени ожидания и если они не получают ответ на свой запрос в течение этого периода, просто повторяют запрос.

Три специфические для SnowFlock оптимизации, проведенные в рамках службы `mcdist`:

- Определение порядка: При временной локализации несколько клонированных виртуальных машин запрашивают одну и ту же страницу в одинаковой последовательности. Сервер `mcdist` игнорирует все запросы за исключением первого.
- Управление потоком: Получатели ответов регулируют частоту приема ответов на запросы. Сервер устанавливает частоту отправки ответов равной средневзвешенной частоте приема ответов клиентами. В противном случае получатели ответов были бы чрезмерно загружены обработкой множества страниц, отправленных сервером.
- Конец игры: Когда сервер отправил большинство страниц, он переходит в режим одноадресной передачи ответов. На данный момент большинство запросов являются повторами, поэтому передача страницы всем узлам не обязательна.

18.5.3. Виртуальный диск

Клонированные с помощью SnowFlock виртуальные машины, ввиду их короткого времени жизни и "детерминизма судьбы", редко используют диск. На виртуальном диске для виртуальной машины SnowFlock размещается корневой раздел с бинарными файлами, библиотеками и файлами конфигурации. Обработка больших объемов данных производится с использованием таких подходящих для этого файловых систем, как [HDFS](#) или PVFS. Следовательно, когда клонированные виртуальные машины SnowFlock принимают решение о необходимости чтения данных с корневого раздела диска, их запросы обычно обслуживаются кэшем страниц файловых систем из состава ядра.

Упомянув об этом, нам все еще необходимо предоставить доступ к виртуальному диску для клонированных виртуальных машин в тех редких случаях, когда он требуется. Мы пошли по пути наименьшего сопротивления и реализовали архитектуру репликации данных диска аналогично архитектуре репликации данных памяти. Во-первых, состояние диска является неизменным во время клонирования. Родительская виртуальная машина продолжает использовать свой диск, осуществляя копирование при записи: данные из записываемых страниц сохраняются в отдельном хранилище и содержимое диска, передаваемое клонированным виртуальным машинам, остается неизменным. Во-вторых, данные состояния диска распространяются с использованием механизма многоадресной передачи между всеми клонированными виртуальными машинами с помощью службы `mcdist`, причем единицей распространения данных является все та же страница размером в 4 КБ и при передаче используются те же механизмы оптимизации на основе временной локализации. В-третьих, репликация данных состояния диска для клонированной виртуальной машины достаточ-

но проста: данные хранятся в обычном файле, который удаляется сразу же после завершения работы клонированной виртуальной машины.

18.6. Компоненты на стороне клонированных виртуальных машин

Клонированные виртуальные машины представляют из себя пустые программные оболочки сразу после создания с помощью архитектурного дескриптора, поэтому как и в случае людей им требуется значительная помощь со стороны родителей во время роста: дочерние виртуальные машины покидают домашнее пространство, но всегда обращаются к нему в случаях, когда обнаруживают факт отсутствия каких-либо необходимых им данных, прося родительские виртуальные машины отправить им эти данные незамедлительно.

18.6.1. Процесс memtap

Присоединяясь к каждой клонированной виртуальной машине после ее создания, процесс `memtap` является жизненно важным для нее программным компонентом. Он отображает всю память клонированной виртуальной машины и заполняет ее данными в случае необходимости. Для реализации некоторых ключевых функций используется гипервизор Xen: права доступа к страницам памяти клонированных виртуальных машин не устанавливаются, поэтому аппаратные ошибки, вызванные первыми попытками доступа к страницам, перенаправляются гипервизором процессу `memtap`.

В простейшем случае процесс `memtap` просто запрашивает страницу, при обращении к которой произошла ошибка, у сервера памяти, но возможны и более сложные сценарии работы. Во-первых, вспомогательные процессы `memtap` используют службу `mcdist`. Это значит, что в любой момент времени любая страница может быть доставлена только из-за того, что она была запрошена другой виртуальной машиной - в этом заключается красота асинхронного распространения данных. Во-вторых, мы позволяем виртуальным машинам SnowFlock быть многопроцессорными виртуальными машинами. В ином случае работа с ними не доставляла бы удовольствия. Это значит, что множество ошибок должно обрабатываться параллельно, возможно даже для одной и той же страницы. В-третьих, в поздних версиях `memtap` вспомогательные процессы могут явно получать данные для группы страниц, которые могут быть доставлены в любом порядке ввиду отсутствия гарантий доставки данных в определенном порядке сервером `mcdist`. Любые из этих факторов ведут к кошмару параллелизма, а нам необходимо учитывать все эти факторы.

Архитектура `memtap` разрабатывалась с учетом главенствующей роли битовой маски наличия страниц. Битовая маска создается и инициализируется в момент обработки архитектурного дескриптора с целью создания клонированной виртуальной машины. Битовая маска является плоским массивом размера, соответствующего допустимому количеству страниц памяти виртуальной машины. Процессоры Intel поддерживают полезные атомарные инструкции для осуществления битовых преобразований: установка значения бита или проверка значения и его установка может происходить гарантированно атомарно по отношению к другим процессорам системы. Это обстоятельство позволяет нам избежать блокировок в большинстве случаев и, таким образом, предоставить доступ к битовой маске различным объектам из различных защищенных областей: гипервизору Xen, процессу `memtap` и самому гостевому ядру клонированной виртуальной машины.

Когда Xen обрабатывает аппаратную ошибку при первом доступе к странице, он использует битовую маску для принятия решения о том, нужно ли взаимодействовать с `memtap`. Он также использует битовую маску для включения в очередь нескольких виртуальных процессоров как зависящих от одной отсутствующей страницы. Процесс `memtap` добавляет страницы в буфер в той последовательности, в которой они прибывают. Когда буфер заполняется или прибывает явно запрошенная страница, работа виртуальной машины приостанавливается и битовая маска используется для удаления из буфера всех доставленных дублей страниц, которые уже присутствовали. После этого все

оставшиеся необходимые страницы копируются в память виртуальной машины и устанавливаются соответствующие биты в битовой маске.

18.6.2. Оптимально работающие клонированные виртуальные машины избегают чрезмерных запросов

Мы только что упомянули о том, что битовая маска наличия страниц доступна для ядра, работающего в клонированной виртуальной машине, а также о том, что для ее модификации не требуется блокировок. Это позволяет клонированным виртуальным машинам использовать мощную "возможность оптимизации": они могут предотвратить запросы страниц, модифицировав битовую маску и указав, что заданные страницы присутствуют. Эта возможность особенно полезна в плане повышения производительности и безопасна в тех случаях, когда страницы полностью перезаписываются перед использованием.

Ситуации, в которых можно избежать запросов страниц, достаточно распространены. Все операции резервирования памяти в ядре (с использованием функций `vmalloc`, `kzalloc`, `get_free_page`, функции пространства пользователя `brk` и других подобных функций) в конце концов осуществляются с помощью механизма резервирования страниц ядра. Обычно резервирование страниц инициируется промежуточными системами резервирования, которые работают с небольшими участками памяти: системой распределения памяти `slab`, системой распределения памяти для процессов пространства пользователя `malloc` из состава `glibc`, и.т.д. Однако, в случаях явного или неявного резервирования памяти одно ключевое семантическое заключение всегда верно: никто не беспокоится о содержимом страницы памяти, так как ее содержимое будет перезаписано произвольным образом. Зачем тогда получать содержимое такой страницы? Для этого нет причин и эмпирический опыт говорит о том, что отказ от получения таких страниц чрезвычайно выгоден.

18.7. Интерфейс приложений для клонирования виртуальных машин

До этого момента мы рассматривали особенности внутренней реализации процесса эффективного клонирования виртуальных машин. Как бы не были интересны описанные системы, нам необходимо обратить внимание на программные компоненты, использующие эти системы: приложения.

18.7.1. Реализация API

Функции клонирования виртуальных машин доступны приложениям посредством простого API `SnowFlock`, схематично изображенного на [Рисунке 18.1](#). Клонирование, в основном, является двухэтапным процессом. Вначале вы осуществляете запрос резервирования ресурсов для клонированных виртуальных машин, при этом, в зависимости от используемых системных политик, может быть зарезервирован объем ресурсов меньший, чем запрошенный. После этого вы сможете использовать зарезервированные ресурсы для клонирования виртуальной машины. Ключевым условием является то, что ваша виртуальная машина должна выполнять одну операцию. Клонирование виртуальных машин применяется в случаях их использования для выполнения единственного приложения, например, поддержки работы веб-сервера или компонента фермы рендеринга. Если вы используете окружение рабочего стола, в котором множество приложений параллельно вызывают функции клонирования виртуальных машин, вы на пути к хаосу.

<code>sf_request_ticket(n)</code>	Запрашивает резервирование системных ресурсов для клонирования n виртуальных машин. Возвращает структуру <code>ticket</code> , описывающую ресурсы для $m \leq n$ клонированных виртуальных машин.
<code>sf_clone(ticket)</code>	Клонирует виртуальную машину, используя структуру <code>ticket</code> , описывающую зарезервированные ресурсы. Возвращает идентификатор клонированной виртуальной машины ID , $0 \leq ID \leq m$.
<code>sf_checkpoint_parent()</code>	Подготавливает неизменную контрольную точку C для родительской виртуальной машины,

	которая может быть использована для клонирования этой машины по прошествии сколь угодно долгого периода времени.
<code>sf_create_clones(C, ticket)</code>	Аналогична функции <code>sf_clone</code> , но использует контрольную точку С. Клонированные виртуальные машины начнут работу с того момента, когда была вызвана соответствующая функция <code>sf_checkpoint_parent()</code> .
<code>sf_exit()</code>	Завершает работу дочерней виртуальной машины ($1 \leq ID \leq m$).
<code>sf_join(ticket)</code>	Блокирует родительскую виртуальную машину ($ID=0$) до того момента, как все дочерние виртуальные машины, описанные в структуре <code>ticket</code> , достигнут вызова <code>sf_exit</code> . В этот момент все дочерние виртуальные машины прекратят свою работу и структура <code>ticket</code> станет недействительной.
<code>sf_kill(ticket)</code>	Используется только для родительской виртуальной машины, делает недействительной структуру <code>ticket</code> и немедленно завершает работу всех дочерних виртуальных машин.

Таблица 18.1: API клонирования виртуальных машин SnowFlock

API просто формирует сообщения и передает их XenStore, интерфейсу на основе разделяемой памяти с низкой пропускной способностью, используемому Xen для передачи управляющих сообщений. Локальный демон SnowFlock (SnowFlock Local Daemon - SFLD) запускает гипервизор и ожидает запросов. Сообщения извлекаются из очереди, после чего выполняются переданные в них команды и отправляются ответы.

Программы могут контролировать процесс клонирования виртуальных машин напрямую с помощью API, доступного для языков программирования C, C++, Python и Java. Сценарии оболочки, связанные с выполнением программы, могут использовать поставляемые инструменты с интерфейсом командной строки вместо данного API. Такие фреймворки для параллельных вычислений, как MPI, могут включать API в свой состав: программы на основе MPI смогут использовать SnowFlock, не располагая фактами об этом и без модификации исходного кода. Балансировщики нагрузки, используемые перед веб-серверами или серверами приложений, могут использовать API для клонирования подконтрольных им серверов.

Локальные демоны SnowFlock управляют исполнением запросов клонирования виртуальных машин. Они создают и отправляют архитектурные дескрипторы, создают клонированные виртуальные машины, запускают серверы диска и памяти и запускают вспомогательные процессы `memtap`. Они представляют собой миниатюрную распределенную систему, ответственную за управление виртуальными машинами в рамках физического кластера.

Локальные демоны SnowFlock отправляют данные о резервировании ресурсов центральному мастер-демону SnowFlock. Мастер-демон SnowFlock просто взаимодействует с соответствующим программным обеспечением для управления кластером. Мы не видим необходимости повторно изобретать колесо в данном случае, поэтому доверяем выполнение задач резервирования ресурсов, установки квот, политик, и.т.д., такому предназначенному для этих целей программному обеспечению, как Sun Grid Engine или Platform EGO.

18.7.2. Необходимые изменения

После клонирования большинство процессов виртуальных машин не информированы о том, что они уже выполняются не в родительской виртуальной машине, а в ее копии. В большинстве случаев данный подход просто работает и не вызывает нареканий. Все-таки главной задачей операционной системы является изоляция приложений от специфических низкоуровневых данных, таких, как сетевая идентичность. Все же, беспрепятственный процесс клонирования требует использования дополнительного набора механизмов. Основной проблемой является управление сетевой идентичностью клонированной виртуальной системы; для того, чтобы избежать конфликтов и путаницы, мы должны произвести незначительные модификации данных в ходе процесса клонирования. Также из-за того, что эти модификации могут повлечь за собой высокоуровневые адаптации, должно быть предусмотрено включение в цепочку обработчиков для того, чтобы пользователь имел возможность настроить выполнение любых необходимых задач, таких, как (пе-

ре)монтирование сетевых файловых систем, которые зависят от сетевой идентичности клонированной виртуальной машины.

Клонированные виртуальные машины появляются в мире, который в большей степени не ожидает их появления. Родительская виртуальная машина является частью сети, наиболее вероятно управляемой DHCP-сервером или любым другим из несметного числа способов, которые используют системные администраторы в своей работе. Вместо предположения о неминуемо негибком сценарии работы, мы поместим родительскую и все клонированные виртуальные машины в их отдельную частную виртуальную сеть. Клонированные виртуальные машины от одного родителя связаны с уникальными идентификаторами и их IP-адреса в этой частной сети автоматически устанавливаются во время клонирования с помощью функции, зависящей от идентификатора. Это гарантирует то, что вмешательство системного администратора не является необходимым, а также никогда не возникнет коллизий IP-адресов.

Смена IP-адресов производится непосредственно с помощью включенной в цепочку обработчиков драйвера виртуальной сети соответствующей функции. При этом мы также используем драйвер для автоматической генерации синтетических ответов DHCP. Таким образом, независимо от вашего выбора дистрибутива, ваш виртуальный сетевой интерфейс позволит передать корректные данные IP гостевой операционной системе, даже в случае перезагрузки.

Для того, чтобы избежать ситуации, при которой различные родительские виртуальные машины проникают в чужие частные виртуальные сети, а также происходят внутренние DDoS-атаки, виртуальные сети для клонированных виртуальных машин разделяются на канальном уровне (уровне 2 OSI). Мы заимствуем диапазон уникальных для организаций MAC-адресов (Ethernet MAC OUI)³ и назначим их клонированным виртуальным машинам. Назначение уникального MAC-адреса будет зависеть от родительской виртуальной машины. Аналогично тому, как IP-адрес виртуальной машины устанавливается в зависимости от ее идентификатора, ее MAC-адрес, не относящийся к полученному диапазону адресов, также зависит от идентификатора. Драйвер виртуальной сети преобразует MAC-адрес виртуальной машины, предполагая его зависимость от идентификатора, и отфильтровывает весь трафик, движущийся в обоих направлениях в частной виртуальной сети с использованием другого диапазона MAC-адресов. Данное разделение трафика эквивалентно разделению, реализуемому при помощи ebtables, но при этом реализуется значительно проще.

Реализация режима обмена данными виртуальных машин друг с другом является замечательной идеей, но ее не достаточно. Иногда мы хотим, чтобы наши клонированные виртуальные машины отвечали на HTTP-запросы из Интернет или подключались к публичным репозиториям данных. Мы снабдим любое множество родительских и клонированных виртуальных машин отдельной виртуальной машиной для маршрутизации. Эта небольшая виртуальная машина работает как межсетевой экран, управляет пропускной способностью канала и осуществляет преобразование сетевых адресов для трафика, направленного от клонированных виртуальных машин в Интернет. Она также ограничивает входящие соединения к родительской виртуальной машине и к портам с известными номерами. Виртуальная машина для маршрутизации не затрачивает большое количество ресурсов, но является единой точкой централизации для сетевого трафика, что может серьезно ограничить масштабируемость. Одни и те же правила функционирования сети могут быть распределены и применены к каждому узлу, на котором работает клонированная виртуальная машина. Мы пока не выпустили этот экспериментальный патч.

Локальные демоны SnowFlock назначают идентификаторы виртуальных машин и передают драйверам виртуальной сети данные, на основе которых они смогут провести настройку: внутренние MAC- и IP-адреса, директивы DHCP, координаты виртуальной машины для маршрутизации, правила фильтрации трафика, и.т.д.

18.8. Заключение

Изменив принцип работы гипервизора Xen и осуществляя отложенную передачу данных состояния виртуальных машин, SnowFlock способен создавать множество работающих клонированных виртуальных машин в течение нескольких секунд. Таким образом, процесс клонирования виртуальных машин при помощи SnowFlock происходит незамедлительно в реальном времени - это повышает удобство использования облачной инфраструктуры, автоматизируя процессы обслуживания кластера и предоставляя приложениям обширные возможности для контроля над ресурсами облака. Также SnowFlock улучшает отзывчивость облака, ускоряя скорость запуска виртуальных машин в 20 раз и повышая производительность большинства созданных виртуальных машин путем копирования используемых операционной системой и приложениями данных кэшей из памяти. Ключевыми возможностями SnowFlock, обуславливающими высокую производительность, являются эвристические методы оптимизации, позволяющие избежать ненужных запросов страниц памяти, и система многоадресного распространения данных, позволяющая совместно получать данные состояния множеству клонированных виртуальных машин. Все, что потребовалось для реализации данного продуманного приложения - это несколько проверенных технологий, ловкость рук и щедрая помощь системы отладки промышленного уровня.

При работе над SnowFlock мы получили два важных урока. Первым уроком является важность обычно недооцениваемого KISS-подхода. Мы ожидали, что реализация технологии снижения интенсивности потока запросов страниц памяти клонированной виртуальной машиной после запуска окажется достаточно сложной. К нашему особенному удивлению, необходимости в ней не обнаружилось. Система работает достаточно хорошо при различных нагрузках с учетом соблюдения единственного принципа: следует доставлять данные памяти тогда, когда они требуются. Другим примером важности простоты реализации является битовая маска наличия страниц памяти. Простая структура данных с применением понятных атомарных семантик доступа значительно упрощает решение ужасной проблемы совместного доступа, которая заключается в соревновании множества виртуальных центральных процессоров за обновление страниц в условиях их асинхронной доставки с помощью многоадресной технологии распространения.

Вторым уроком была важная роль масштабирования. Другими словами, будьте готовы к тому, что работа вашей системы будет нарушена и новые узкие места будут обнаруживаться каждый раз, когда вы увеличиваете нагрузку на систему в два раза. Данный урок тесно связан с предыдущим: простые и элегантные решения отлично масштабируются и не таят в себе нежелательных сюрпризов и повышений нагрузок. Основным примером в данном случае является система `mcdist`. При тестировании масштабирования в большом диапазоне механизм распространения страниц с использованием протокола TCP/IP в значительной степени замедляет работу при обслуживании сотен клонированных виртуальных машин. Система `mcdist` успешна благодаря очень проработанному и точно обозначенному распределению задач: клиенты беспокоятся только о доставке своих страниц; сервер беспокоится только о глобальном управлении потоком. Поддержка `mcdist` в виде небольшого и простого приложения, обуславливает отличную возможность масштабирования SnowFlock.

Если вам хочется узнать больше, вы можете посетить сайт Университета Торонто¹, чтобы ознакомиться с академическими отчетами и открытым под лицензией GPLv2 исходным кодом, а также сайт GridCentric⁴ для получения реализации промышленного уровня.

Сноски

1. <http://sysweb.cs.toronto.edu/projects/1>
2. <http://www.gridcentriclabs.com/architecture-of-open-source-applications>
3. Уникальный идентификатор организации (OUI, Organizational Unique ID) является диапазоном MAC-адресов, закрепленным за производителем оборудования.
4. <http://www.gridcentriclabs.com/>

19. Электронные таблицы SocialCalc

История электронных таблиц длится более 30 лет. Первая программа электронных таблиц, VisiCalc, была придумана Дэном Бриклином (Dan Bricklin) в 1978 году и выпущена в 1979 году. Первоначальная концепция была довольно проста: таблица, которая имеет бесконечные размеры в двух направлениях, ячейки которой заполняются текстом, цифрами и формулами. Формулы состоят из обычных арифметических операторов и различных встроенных функций, причем каждая формула может использовать в качестве значений текущее содержимое других ячеек.

Хотя метафора была простой, у нее было много применений: бухгалтерский учет, учет товаров и управление различными списками являются лишь некоторыми из них. Возможности практически безграничны. Все эти варианты применения сделали VisiCalc первым «крутым приложением» эры персональных компьютеров.

В последующие десятилетия, как появились преемники, например, Lotus 1-2-3 и Excel, были сделаны постепенные улучшения, но основная метафора осталась прежней. Большинство таблиц сокращались в файлах, расположенных на диске, и загружались в память, когда они открывались. Совместная работа была особенно трудна из-за лежащей в основе файловой модели:

- Каждый пользователь должен был установить версию редактора электронных таблиц.
- Обмен электронными письмами, общие папки или создание специальных систем контроля версий - все это увеличивало бухгалтерские накладные расходы.
- Возможность отслеживания изменений была ограничена; например, Excel не сохраняет историю изменения форматов и комментариев к ячейкам.
- Изменение форматов или формул в шаблонах требовало кропотливых изменений в существующие файлах электронных таблиц, которые использовали эти шаблоны.

К счастью, для решения этих вопросов с элегантной простотой появилась новая модель совместной работы. Это модель wiki, изобретенная Уордом Каннингемом (Ward Cunningham) в 1994 году и популяризированная Википедией в начале 2000-х годов.

Вместо файлов, модель wiki использует страницы, размещенные на сервере и редактируемые в браузере, не требуя для этого специального программного обеспечения. Эти гипертекстовые страницы могут легко быть связаны друг с другом, и даже включать в себя части других страниц, формируя большие страницы. По умолчанию все участники просматривают и редактируют последнюю версию, а слежение за историей документа осуществляется с помощью сервера.

Вдохновленный моделью wiki, Дэн Бриклин начал работать над WikiCalc в 2005 году. Его целью было объединить простоту создания и возможность совместного редактирования несколькими пользователями страниц wiki со знакомой метафорой визуального форматирования и вычисления электронных таблиц.

19.1. WikiCalc

Первая версия WikiCalc (рис. 19.1) имела несколько особенностей, которые в то время отличали ее от других таблиц:

- Для отображения текстовых данных использовался обычный текст, текст на HTML и разметка в стиле wiki.
- Тест в стиле wiki, который включал в себя команды для вставки ссылок, изображений и значений из ячеек, на которых делались ссылки.
- Ячейки с формулами могли ссылаться на значения на других страницах WikiCalc, размещенных на других сайтах.
- Возможность создавать результат, который можно было добавлять в другие веб-страницы, причем как статические, так и динамические данные.
- Форматирование ячеек, использующее атрибуты стилей CSS и классы CSS.

- Журналирование всех операций редактирования для последующего аудита.
- Сохранение каждой новой версии страницы средствами wiki с возможностью отката.

The screenshot shows a window titled "PAGE SELECTION". At the top is a menu bar with "Page", "Edit", "Format", "Publish", "Tools", and "Quit". Below the menu is a section titled "PAGE SELECTION" with the text: "This is where you choose which page you want to edit. You can also change which site you are editing. Open a page for editing by pressing the appropriate Edit button. It will be copied from the server and you will be editing that copy. Modified pages may be published (which updates the copy on the server) and editing closed by pressing the appropriate Publish button." Below this is a section titled "Pages You Can Edit On Site: Site setup by Demonstration Setup (demosite)". It displays the message "Your author name is: demoauthor". Underneath are three radio buttons: "Edit Buttons" (selected), "View On Web Buttons", and "Delete and Abandon Edit Buttons". A table follows, listing three files with their edit and publish status:

	FILENAME	FULL NAME	EDIT STATUS	PUBLISH STATUS
	ax.html	axax	Currently being edited Last modified: Apr 24, 2011 07:10:48	[Not Published]
	demopage1.html	wikiCalc Demonstration Page	Open for editing Not modified	[Not Published]
	fil.html	bar	Open for editing Last modified: Apr 24, 2011 07:10:48	[Not Published]

Рис.19.1: Интерфейс WikiCalc 1.0

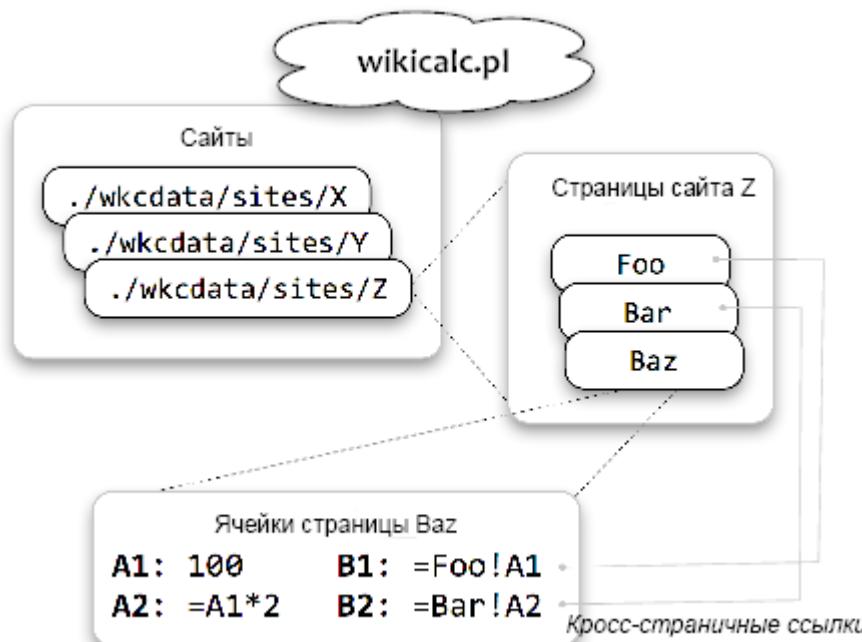


Рис.19.2: Компоненты WikiCalc

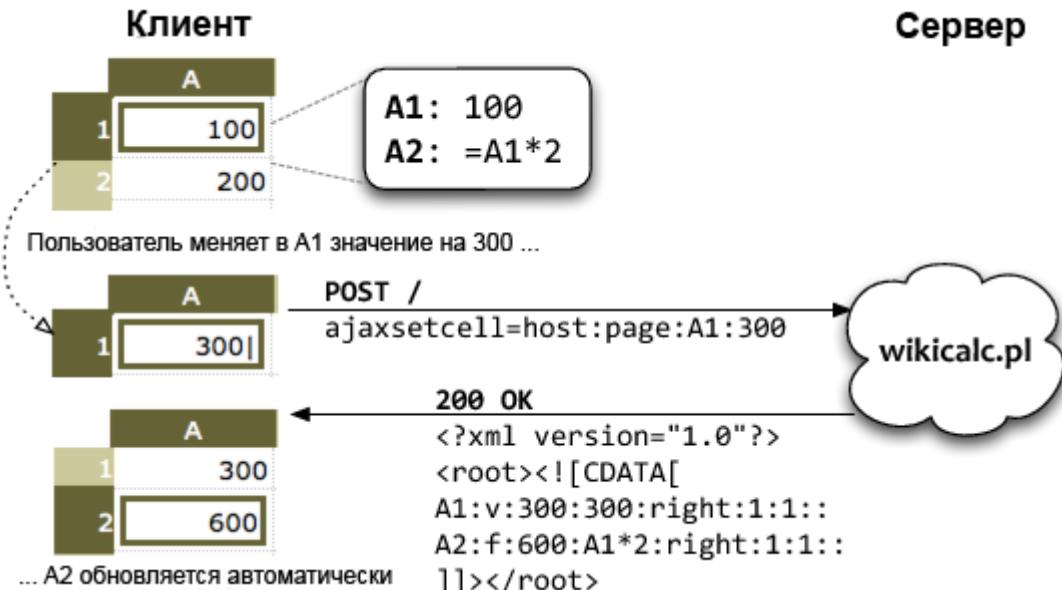


Рис.19.3: Поток данных в WikiCalc

Внутренняя архитектура (рис.19.2) и информационные потоки (рис.19.3) были преднамеренно простыми, но, тем не менее, мощными. Возможность составлять мастер таблицу из нескольких меньших таблиц оказалась особенно удобной. Например, представьте себе ситуацию, когда каждый продавец хранит показатели продаж на странице электронной таблицы. Затем каждый менеджер сводит их показатели в региональную таблицу, а вице-президент по продажам затем сводит региональные показатели в таблицу верхнего уровня.

Каждый раз, когда одна из отдельных таблиц обновляется, все таблицы более высокого уровня могут отобразить обновление. Если для кого-нибудь нужны подробности, то нужно будет просто щелкнуть мышкой и посмотреть в таблицу, лежащую за текущей таблицей. Такая возможность сведения воедино данных исключает необходимость вносить изменения в несколько мест, что может быть причиной ошибок, и гарантирует, что вся отображаемая информация будет самой свежей.

Для того, чтобы все перерасчеты оставались актуальными, в WikiCalc была использована концепция тонкого клиента, когда хранение всей информации осуществляется на серверной стороне. Каждая электронная таблица представлена в браузере в виде элемента `<table>`; при редактировании ячейки на сервер отсылается вызов `ajaxsetcell`, а сервер затем сообщает браузеру, какие ячейки нуждаются в обновлении.

Неудивительно, что такое решение зависито от скорости соединения между браузером и сервером. Когда задержка высокая, пользователи начинают обнаруживать между началом обновления ячейки и ее новым содержимым частое появление сообщения «Loading ...» («Загрузка ...») так, как это показано на рис.19.4. Это особенно актуально для пользователей, интерактивно редактирующих формулы при помощи настройки входных данных и ожидающих увидеть результаты в реальном времени.

	A	B	C	D
1	Loading...			
2				
3	Sample financial calculation in a table with borders	Year	2006	2007
4		Sales	Loading...	170.5
5		Cost	124.0	136.4
6		Profit	31.0	34.1

Рис.19.4: Сообщение о загрузке

Кроме того, поскольку элемент `<table>` имеет те же самые размеры, что и электронная таблица, для сетки размером 100×100 создается 10000 DOM-объектов `<tdd>`, которые отьедают ресурсы памяти браузеров и еще более ограничивая размеры страниц.

Из-за этих недостатков, хотя WikiCalc был полезным в качестве автономного работающего сервера, работающего в локальной сети, он на практике был не очень удобен для того, чтобы внедрять его как часть веб-систем управления контентом.

В 2006 году Дэн Бриклин объединился с компанией Socialtext с тем, чтобы начать разработку проекта SocialCalc, варианта WikiCalc, частично переработанного с Javascript в некоторое подмножество кода на языке Perl.

Такая переработка была предназначена поддержки возможности ведения совместных работ в большем объеме и изменения внешнего вида электронный таблиц таким образом, чтобы они стали похожи на настольные приложения. Другими целями проекта были следующие:

- Возможность обработки сотен тысяч ячеек.
- Более быстрое выполнение операций редактирования.
- Возможность на клиентской стороне осуществлять аудит и выполнять операции undo/redo.
- Более эффективное использование Javascript и CSS с тем, чтобы более полноценные функциональные возможности компоновки.
- Кросс-браузерная поддержка, несмотря на более широкое использование языка Javascript.

После трех лет разработки и различных бета-версий, компания Socialtext в 2009 году выпустила SocialCalc 1.0, успешно реализовав цели разработки. Давайте теперь взглянем на архитектуру системы SocialCalc.

19.2. SocialCalc

The screenshot shows the SocialCalc interface. At the top, there is a toolbar with buttons for Save, Preview, Refresh, Cancel, and a file name 'test'. Below the toolbar is a menu bar with icons for file operations like New, Open, Save, Print, and a Format dropdown. The main area is a spreadsheet grid with columns labeled A, B, C, D, E and rows labeled 1, 2, 3, 4. Cell A1 contains the value '1874', cell B1 contains '172', and cell A3 contains '2046'. The cell containing '2046' has a blue selection border. The status bar at the bottom shows the number '1874'.

	A	B	C	D	E
1	1874				
2	172				
3	2046				
4					

Рис.19.5. Интерфейс SocialCalc

На рис.19.5 и рис.19.6 показаны интерфейс и классы SocialCalc, соответственно. В сравнении с WikiCalc, роль сервера была значительно сокращена. Его единственной обязанностью осталось отвечать на запросы HTTP GET и обслуживать все электронные таблицы, сериализуя их в формате save, позволяющим их сохранять; как только браузер получал данные, все расчеты, отслеживание изменений и взаимодействия пользователем теперь осуществлялись на языке Javascript.

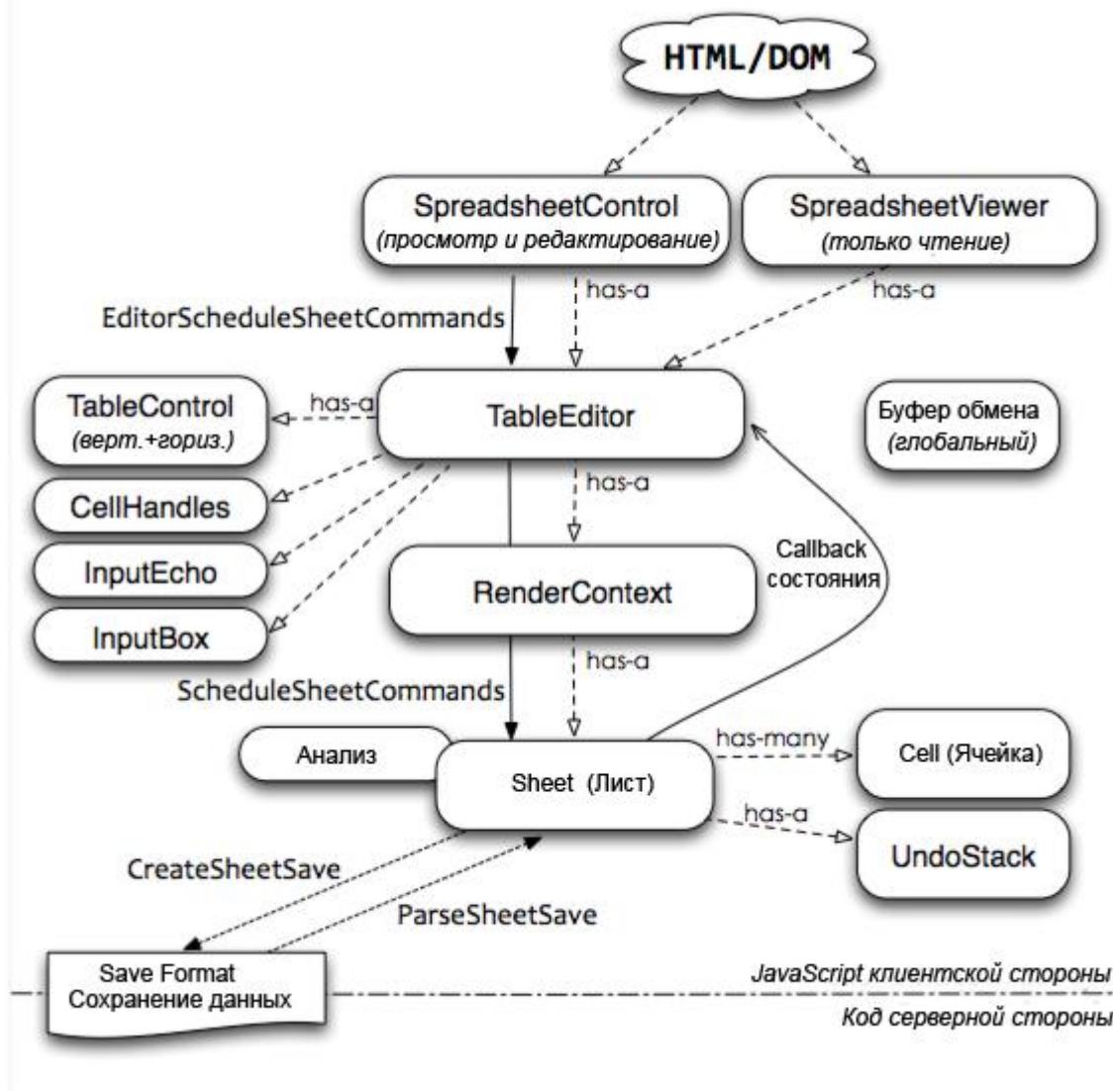


Рис.19.6: Диаграмма классов SocialCalc

Компоненты Javascript были разработаны в стиле послойной модели MVC (Model/View/Controller — Модель/Внешний вид/Контроллер), причем задача каждого класса фокусировалась на одном аспекте:

- Класс *Sheet* является моделью данных, представляющих структуру электронной таблицы, размещенную в памяти. В ней хранится словарь из координат для объектов *Cell* (Ячейка). Он содержит словарь из координат сотовых объектов, каждый из которых представляет одну ячейку. В пустых ячейках нет записей и, поэтому, для них вообще память не тратится.
- Класс *Cell* представляет собой содержимое и формат ячейки. Некоторые обычные свойства приведены в таблице 19.1.
- Класс *RenderContext* реализует компонент отображения, на него возложена обязанность отображать лист в объекты DOM.
- Класс *TableControl* является основным контроллером, принимающим события мыши и клавиатуры. Как только он получит событие, связанное с отображением, например, прокруткой и изменением размеров, он обновляет связанный с ними объект *RenderContext*. Как только он получит обновления события, которые влияют на содержимое листа, он создает новые команды, которые помещаются в очередь команд листа.
- Класс *SpreadSheetControl* является пользовательским интерфейсом верхнего уровня с панелями инструментов, строками состояний, диалоговыми окнами и палитрами для выбора цвета.
- Класс *SpreadSheetViewer* является альтернативным пользовательским интерфейсом верхнего уровня, который предназначен только для интерактивного просмотра.

Таблица 19.1: Содержимое и форматы ячеек

datatype	t
datavalue	1Q84
color	black
bgcolor	white
font	italic bold 12pt Ubuntu
comment	Ichi-Kyu-Hachi-Yon

Мы взяли на вооружение систему объектов с минимальным количеством классов и простыми механизмами композиции/делегирования и не использовали наследование или прототипы объектов. Все символы помещаются в пространство имен `SocialCalc.*` с тем, чтобы избежать конфликта имен.

Каждое обновление на листе проходит через метод `ScheduleSheetCommands`, который принимает строку, представляющую собой команду редактирования. Некоторые обычные команды показаны в таблице 19.2. В приложении, в которое встроен `SocialCalc`, можно самостоятельно определять дополнительные команды при помощи добавления именованных обратных вызовов в объект `SocialCalc.SheetCommandInfo.CmdExtensionCallbacks` и использования команды `startcmdextension` для обращения к ним.

Таблица 19.2: Команды SocialCalc

set sheet defaultcolor blue	erase A2
set A width 100	cut A3
set A1 value n 42	paste A4
set A2 text t Hello	copy A5
set A3 formula A1*2	sort A1:B9 A up B down
set A4 empty	name define Foo A1:A5
set A5 bgcolor green	name desc Foo Used in formulas like SUM(Foo)
merge A1:B2	name delete Foo
unmerge A1	startcmdextension UserDefined args

19.3. Цикл работы команд

Чтобы улучшить ответную реакцию, `SocialCalc` выполняет все перерасчеты и обновления DOM в фоновом режиме, так что пользователь может вносить новые изменения в несколько ячеек в то время, как движок выполняет более ранние изменения, записанные в очереди команд.

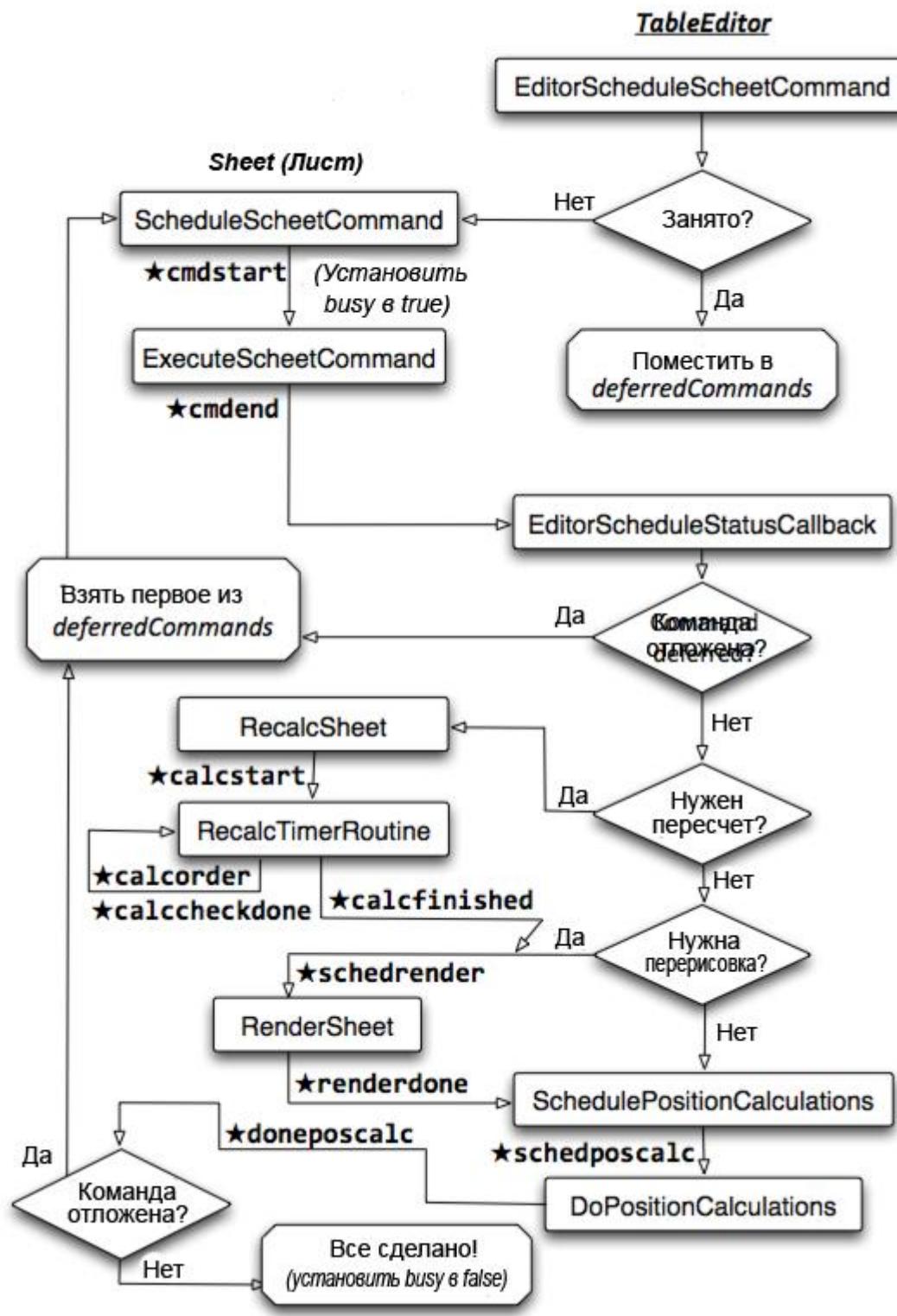


Рис.19.7: Цикл работы команд SocialCalc

Когда выполняется команда, объект `TableEditor` устанавливает свой флаг `busy` (занято) в состояние `true` (истина); затем последующие команды помещаются в очередь `deferredCommands`, что обеспечивает последовательный порядок выполнения команд. Как видно на диаграмме цикла событий, показанной на рис.19.7, объект `Sheet` продолжает посыпать события `statusCallback` с тем, чтобы уведомить пользователя о текущем состоянии выполнения команды, которая может находиться одном из следующих четырех состояний:

- *ExecuteCommand* (выполнение команды): Посыпает сигнал `cmdstart` при старте и сигнал `cmdend`, когда команда завершает выполнение. Если команда косвенно изменила значение ячейки, то переходим к щану *Recalc*. Иначе, если команда изменила внешний вид одной

или нескольких ячеек, отображаемых на экране, то переходим к шагу *Render*. Если не произошло ни одно из вышеперечисленных условий (например, в случае команды копирования `copy`), то происходит переход к шагу *PositionCalculations*.

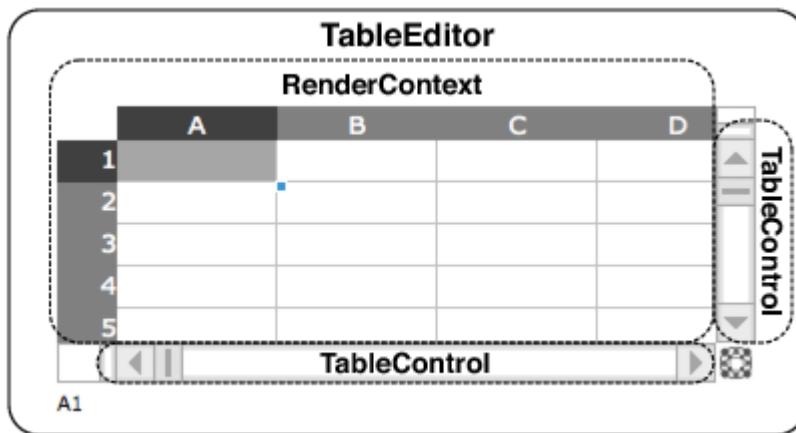
- *Recalc* (при необходимости) (перерасчет): Посыпает сигнал `calcstart` при старте, сигнал `calcorder` каждые 100 мс при проверке цепочки зависимостей ячеек, сигнал `calccheckdone` когда проверка оканчивается, и сигнал `calcfinished`, когда все соответствующие ячейки получат свои перевычисленные значениями. За эти шагом всегда следует шаг *Render*.
- *Render* (при необходимости) (Отображение): Посыпает сигнал `schedrender` и сигнал `renderdone` когда элемент `<table>` изменяется с изменением формата ячеек. За этим шагом всегда следует шаг *PositionCalculations*.
- *PositionCalculations* (вычисление положения): Посыпает сигнал `schedposcalc` при запуске и сигнал `doneposcalc` после обновления полос прокрутки, курсора текущей редактируемой ячейки и других визуальных компонентов `TableEditor`.

Поскольку все команды будут сохранены после того, как они будут выполнены, мы естественным образом получаем журнал аудита всех операций. Метод `Sheet.CreateAuditString` добавляет в журнал аудита символы новой строки с тем, чтобы каждая команда была в отдельной строке.

`ExecuteSheetCommand` также для каждой команды, которую он выполняет, создает команду отмены `undo`. Например, если в ячейке A1 находится текст «Foo» и пользователь выполняет команду `set A1 text Bar`, то в стек команд отмены будет помещена команда `set A1 text Foo`. Если пользователь нажимает кнопку `Undo`, то будет выполнена команда `undo`, которая восстановит содержимое ячейки A1 к исходному значению.

19.4. Редактор таблиц

Теперь давайте посмотрим на слой `TableEditor`. Он с помощью своего метода `RenderContext` рассчитывает экранные координаты и с помощью двух экземпляров метода `TableControl` управляет горизонтальной/вертикальной прокруткой.



Управление прокруткой с помощью экземпляров метода `TableControl`

Слой отображения, обрабатываемый классом `RenderContext`, также отличается от конструкции `WikiCalc`. Вместо отображения каждой ячейки в элемент `<td>`, мы теперь просто создаем элемент `<table>` фиксированного размера, который соответствует видимой области браузера и заранее заполняем ее элементами `<td>`.

Когда пользователь выполняет прокрутку с помощью наших полос прокрутки, мы динамически обновляем `innerHTML` предварительного нарисованных элементами `<td>`. Это означает, что нам

не нужно в большинство случаев создавать или уничтожать какие-либо элементы `<tr>` и `<td>`, что существенно ускоряет время отклика.

Поскольку `RenderContext` отрисовывает только видимую область, размер объекта `Sheet` (Лист) может быть сколь угодно большой, что не влияет на производительность.

В `TableEditor` также есть объект `CellHandles`, в котором реализовано круговое меню `fill/move/slide` (заполнение/перемещение/сдвиг), прикрепленное к правому нижнему углу ячейки, редактируемой в текущий момент, которая известна как `Ecell` и которая показана на рис.19.9.

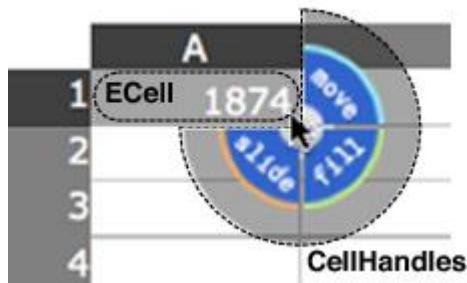


Рис.19.9: Ячейка, редактируемая в текущий момент и называющаяся Ecell

Управление полем ввода осуществляется двумя классами: `InputBox` и `InputEcho`. Первый из них управляет строкой редактирования, расположенной над сеткой, а второй, который показан как слой просмотра, обновляемый по мере того, как вы вводите текст, выполняет наложение содержимого `ECell` (рис.19.10).

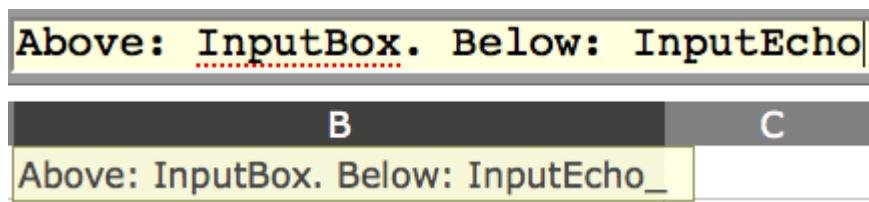


Рис.19.10: Поле ввода, управляемое двумя классами

Обычно движок `SocialCalc` нужен только для связи с сервером в случае, когда электронная таблица открывается для редактирования и когда отправляет сохраняемые данные на сервер. Для этой цели используется метод `Sheet.ParseSheetSave`, который анализирует формат `save`, используемый для сохранения данных, и преобразует данные в объект `Sheet`, и метод `Sheet.CreateSheetSave`, сериализующий объект `Sheet` обратно в формат `save`.

В формулах можно с помощью адресов URL делать ссылки на значения в любой таблице, расположенной удаленно. Команда `recalc` пересчитывает электронные таблицы, в которых есть внешние ссылки, затем анализирует их с помощью метода `Sheet.ParseSheetSave` и сохраняет их в кэше, так что пользователь может делать ссылки на другие ячейки в той же самой таблице, расположенной удаленно, без повторного извлечения ее содержимого.

19.5. Формат сохранения save

Формат сохранения `save` является стандартным форматом MIME `multipart/mixed`, состоящим из четырех частей `ext/plain; charset=UTF-8`; в каждой части находится текст, разделяемый символами новой строки, с полями, которые отделяются друг от друга с помощью двоеточий. Это следующие части:

- Часть `meta`, в которой перечислены типы других частей.
- Часть `sheet`, в котором перечислены формат и содержимое каждой ячейки, ширина каждого столбца (если не используется значение, определяемое по умолчанию), за которым следует список шрифтов, вариантов цвета и границ, используемых в листе.
- Необязательная часть `edit`, сохраняющая состояние редактора `TableEditor`, в том числе последнее положение элемента `Ecell`, а также зафиксированные размеры панелей строк/столбцов.
- Необязательная часть `audit`, в которой хранится история команд, выполненных в предыдущем сеансе редактирования.

Например, на рис.19.11 показана таблица с тремя ячейками со значением 1874 в ячейке A1, представляющей собой элемент ECell, формулой 2^{2*43} в ячейке A2, и формулой `SUM(Foo)` в ячейке A3, выделенные жирным шрифтом, и ссылкой на диапазон A1:A2 с именем `Foo`.

A	
1	1874
2	"Foo"
3	172 = 2^{2*43}
	2046 = <code>SUM(Foo)</code>

Рис.19.11: Таблица с тремя ячейками

Сериализованный формат save таблицы выглядит следующим образом:

```

socialcalc:version:1.0
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=SocialCalcSpreadsheetControlSave
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

# SocialCalc Spreadsheet Control Save
version:1.0
part:sheet
part:edit
part:audit
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

version:1.5
cell:A1:v:1874
cell:A2:vtf:n:172:2^2*43
cell:A3:vtf:n:2046:SUM(Foo):f:1
sheet:c:1:r:3
font:1:normal bold * *
name:FOO::A1\cA2
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

version:1.0
rowpane:0:1:14
colpane:0:1:16
ecline:A1
--SocialCalcSpreadsheetControlSave
Content-type: text/plain; charset=UTF-8

set A1 value n 1874
set A2 formula 2^2*43
name define Foo A1:A2
set A3 formula SUM(Foo)

```

--SocialCalcSpreadsheetControlSave--

Этот формат создавался таким образом, чтобы он был понятен человеку и чтобы его можно было относительно легко генерировать программно. Это дает возможность с помощью плагина Sheetnode из фреймворка Drupal, использующего язык PHP, выполнять преобразования между этим форматом и другими популярными форматами электронных таблиц, например, Excel (.xls) и OpenDocument (.ods).

Теперь, когда мы понимаем, как отдельные части в SocialCalc, сочетаются друг с другом, давайте рассмотрим два реальных примера расширений SocialCalc.

19.6. Расширенные возможности редактирования

Первый пример, рассматриваемый нами, является улучшенным вариантом текстовой ячейки SocialCalc, которая позволяет непосредственно в редакторе таблиц отображать текст со расширенными возможностями (рис. 19.12).

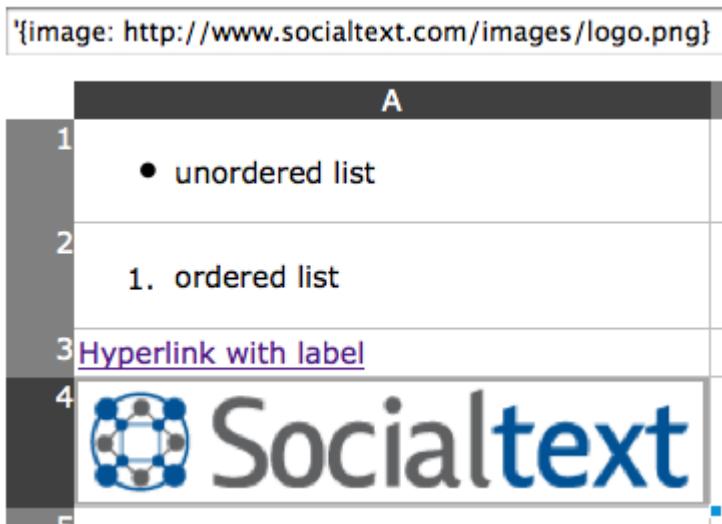


Рис.19.12: Отображение в табличном редакторе текста с расширенными возможностями редактирования

Мы добавили эту возможность в SocialCalc сразу после его релиза 1.0 для того, чтобы выполнить популярную просьбу разрешить в рамках единого синтаксиса вставлять изображения, ссылки и текстовую разметку. Поскольку Socialtext уже имел wiki-платформу с открытым исходным кодом, было естественным повторно использовать синтаксис также и для SocialCalc.

Чтобы осуществить это, нам нужно специальным образом отрисовывать `textvalueformat` из `text-wiki` и, чтобы использовать этот результат, нужно изменить формат текстовых ячеек, определяемый по умолчанию.

Вы спросите, а что такое этот формат `textvalueformat`? Читаем дальше.

19.6.1. Типы и форматы

В SocialCalc, каждая ячейка имеет тип данных `datatype` и тип значений `valuetype`. Ячейки данных с текстом или цифрами соответствуют типам значений `text/numeric` (текстовый/числовой), а ячейки с формулами типа `datatype="f"` могут генерировать либо числовые, либо текстовые значения.

Вспомним, что на шаге визуализации Render, объект Sheet генерирует HTML для каждой из своих ячеек. Он делает это проверяя тип значений `valuetype` каждой ячейки: если он начинается с `t`, то атрибут ячейки `textvalueformat` определяет, как должна выполняться генерация. Если он начинается с `n`, то вместо этого используется атрибут `nontextvalueformat`.

Однако, если атрибут ячейки `textvalueformat` или `nontextvalueformat` не определен явно, то формат, используемый по умолчанию, берется из `valuetype` так, как это показано на рис.19.13.

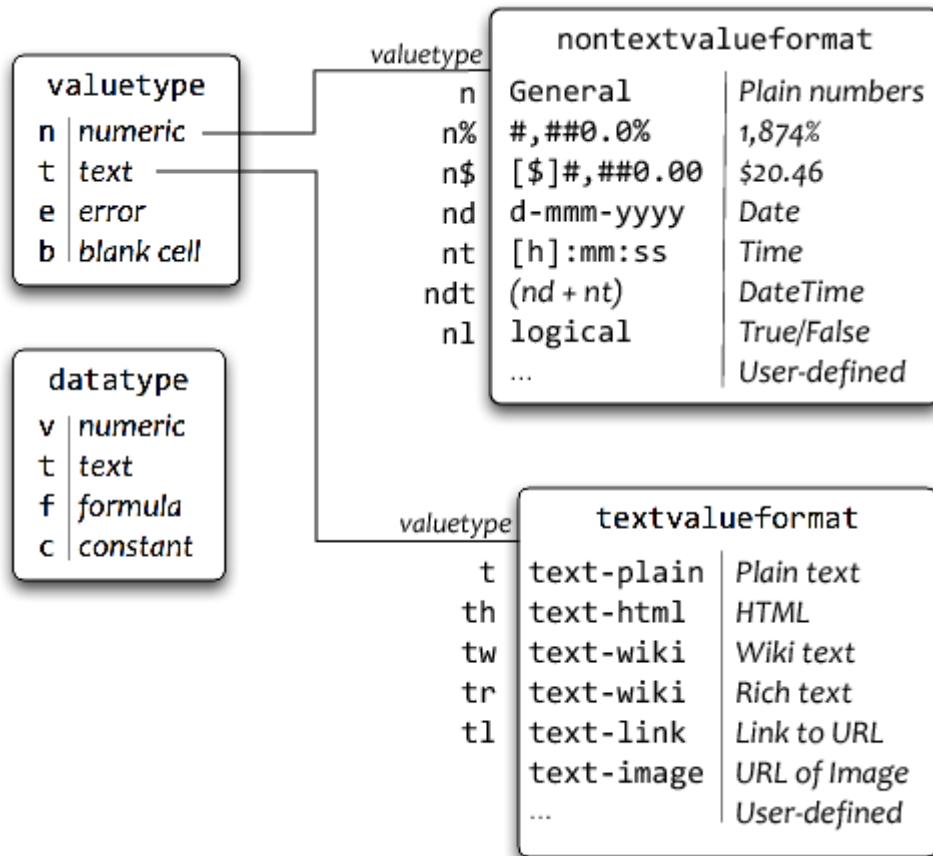


Рис.19.13: Типы значений

Поддержка формата значений `text-wiki` закодирована в `SocialCalc.format_text_for_display` следующим образом:

```

if (SocialCalc.Callbacks.expand_wiki && /^text-wiki/.test(valueformat)) {
    // do general wiki markup
    displayvalue = SocialCalc.Callbacks.expand_wiki(
        displayvalue, sheetobj, linkstyle, valueformat
    );
}

```

Вместо того, чтобы непосредственно вставлять расширение wiki-to-HTML (преобразующее wiki-текст в HTML — прим.пер.) в `format_text_for_display`, мы определим новый триггер в `SocialCalc.Callbacks`. Этот стиль рекомендуется использовать везде в коде `SocialCalc`; он улучшает модульность благодаря тому, что позволяет различными способами подключать расширения для wiki-текста, а также сохранять совместимость с уже встроенными возможностями, в которых этот прием не нужен.

19.6.2. Отображение wiki-текста

Затем мы будем использовать Wikiwyg [1], библиотеку Javascript, осуществляющую двунаправленное преобразование между wiki-текстом и HTML.

Мы определяем функцию `expand_wiki`, которая берет из ячейки текст, пропускает его через анализатор wiki-текста в Wikiwyg и выдает текст на языке HTML:

```
var parser = new Document.Parser.Wikitext();
var emitter = new DocumentEmitter.HTML();
SocialCalc.Callbacks.expand_wiki = function(val) {
    // Convert val from Wikitext to HTML
    return parser.parse(val, emitter);
}
```

Последний шаг включает в себя выполнение команды `text-wiki - set sheet defaulttextvalueformat` сразу после инициализации таблицы:

```
// Мы допускаем, что в DOM уже имеется
```

```
var spreadsheet = new SocialCalc.SpreadsheetControl();
spreadsheet.InitializeSpreadsheetControl("tableeditor", 0, 0, 0);
spreadsheet.ExecuteCommand('set sheet defaulttextvalueformat text-wiki');
```

Если все объединит вместе, то шаг визуализации Render теперь работает так, как это показано на рис.19.14.

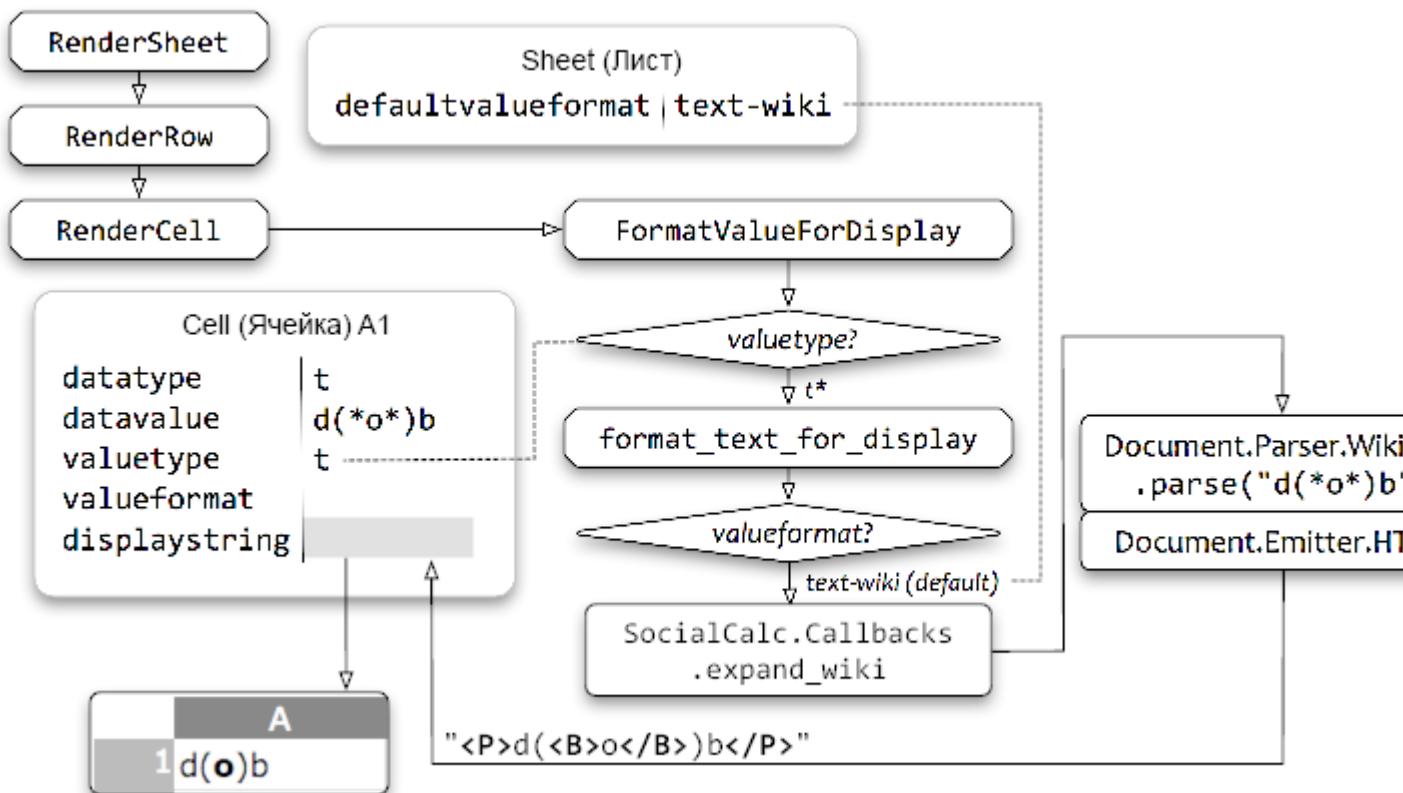


Рис.19.14: Шаг визуализации Render

Вот и все! Расширенный вариант SocialCalc теперь поддерживает расширенный набор синтаксиса, позволяющего использовать wiki-разметку:

```
*bold* _italic_ `monospace` {{unformatted}}
> indented text
* unordered list
```

```
# ordered list
"Hyperlink with label"
{image: http://www.socialtext.com/images/logo.png}
```

Попробуйте ввести в ячейку A1 текст ***bold*** *italic* `monospace`, и вы увидите, что он отображается в уже отформатированном виде (рис.19.15).

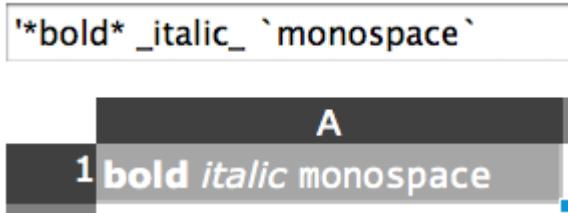


Рис.19.15: Пример Wikwyg

19.7. Совместные работы в режиме реального времени

Следующий пример, который мы изучим, является многопользовательской общедоступной электронной таблицей, редактируемой в режиме реального времени. Это на первый взгляд может показаться сложным, но благодаря модульной конструкции SocialCalc все, что нужно для каждого он-лайн пользователя, это транслировать свои команды другим участникам работы.

Чтобы различать локально-используемые команды и команды, используемые дистанционно, мы в метод ScheduleSheetCommands добавляем параметр `isRemote`:

```
SocialCalc.ScheduleSheetCommands = function(sheet, cmdstr, saveundo, isRemote) {
    if (SocialCalc.Callbacks.broadcast && !isRemote) {
        SocialCalc.Callbacks.broadcast('execute', {
            cmdstr: cmdstr, saveundo: saveundo
        });
    }
    // ... здесь собственно код ScheduleSheetCommands here...
}
```

Теперь все, что нам нужно сделать, это определить подходящую функцию обратного вызова `SocialCalc.Callbacks.broadcast`. Как только она появится, у всех пользователей, подключенных к той же самой таблице, будут выполняться одни и те же команды.

Когда эта функция была впервые реализована для OLPC (One Laptop Per Child — Проект «Каждому ребенку свой ноутбук» [2]) лабораторией Sugar Labs в Сита (Уганда) [3] в 2009 году, то функция `broadcast` была создана с использованием вызовов XPCOM в D-Bus/Telepathy - стандартного транспорта для сетей OLPC/Sugar (смотрите рис.19.16).

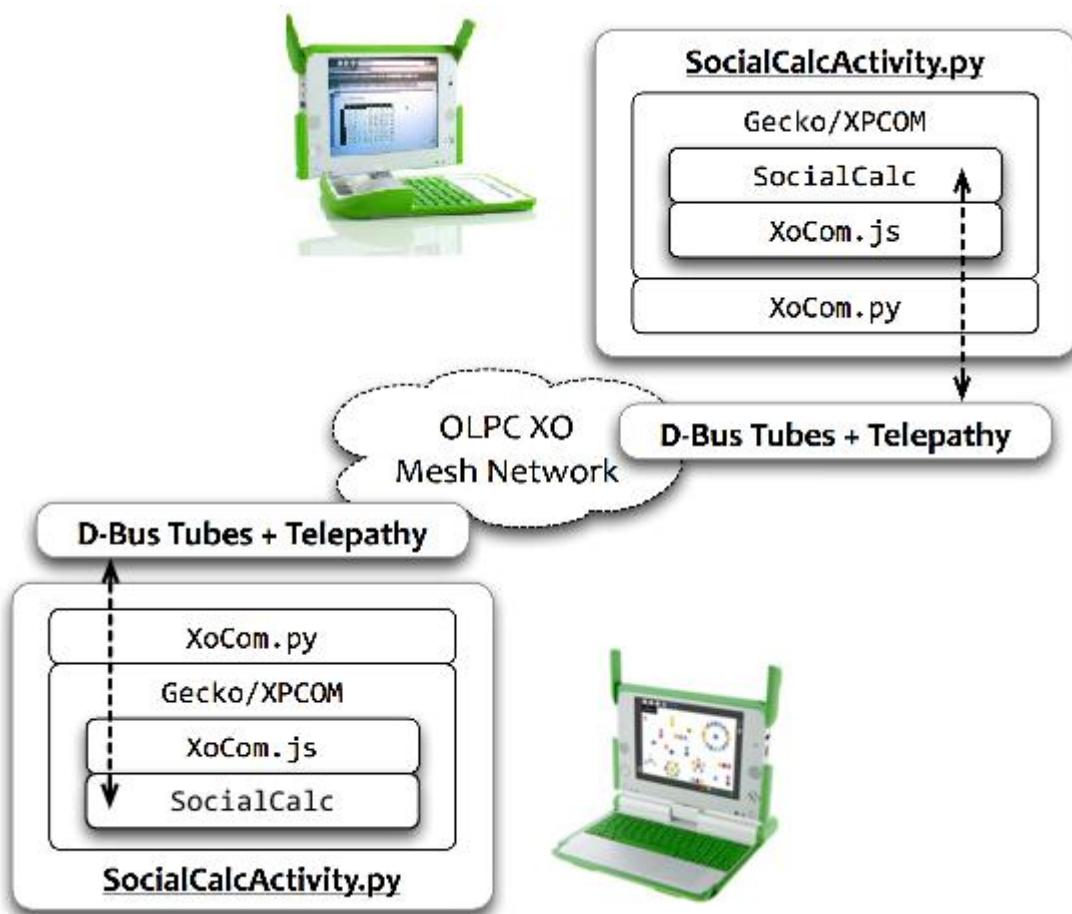


Рис.19.16: Реализация OLPC

Это работает достаточно хорошо, позволяя в одной и той же сети Sugar использовать экземпляры объектов XO для совместной работы над общей таблицей SocialCalc. Однако, есть особенности, касающиеся как браузерной платформы Mozilla/XPCOM, так и платформы обмена сообщениями D-Bus/Telepathy.

19.7.1. Кросс-браузерный транспорт

Чтобы можно было выполнять эту работу в разных браузерах и разных операционных системах, мы пользуемся фреймворком `Web::Hippie` [4], высокогоуровневой абстракции JSON-поверх-WebSocket с удобной привязкой к Jquery и с MXHR (запроса XML HTTP, состоящего из нескольких частей [5]) в качестве резервного механизма транспорта на случай, если WebSocket недоступен.

Для браузеров с установленным плагином Adobe Flash, но без встроенной поддержки WebSocket, мы используем Flash-эмulation WebSocket из проекта `web_socket.js` [6], которая работает даже быстрее и надежнее, чем MXHR. Поток операций показан на рис.19.17.

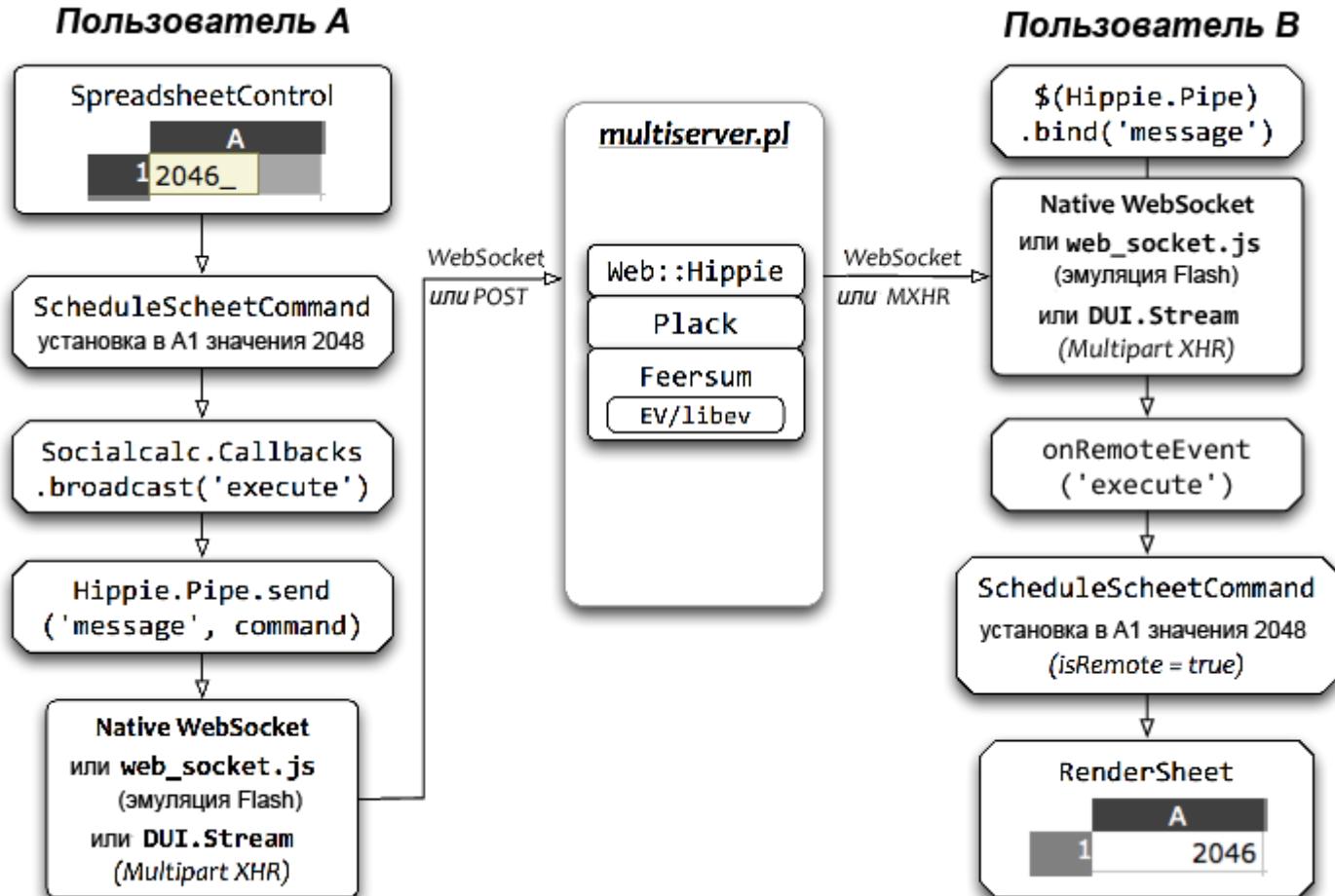


Рис.19.17: Кросс-браузерный поток операций

Функция клиентской стороны `SocialCalc.Callbacks.broadcast` определяется следующим образом:

```
var hpipe = new Hippie.Pipe();

SocialCalc.Callbacks.broadcast = function(type, data) {
    hpipe.send({ type: type, data: data });
};

$(hpipe).bind("message.execute", function (e, d) {
    var sheet = SocialCalc.CurrentSpreadsheetControlObject.context.sheetobj;
    sheet.ScheduleSheetCommands(
        d.data.cmdstr, d.data.saveundo, true // isRemote = true
    );
    break;
});
```

Хотя это работает достаточно хорошо, есть еще два оставшихся вопроса, которые нужно решить.

19.7.2. Разрешение конфликтов

Первым вопросом является состояние гонки (race-condition) во время выполнения команд: Если пользователи А и В одновременно выполняют операции, влияющие на одни и те же ячейки, принимают и выполняют команды, транслируемые другими пользователями, то в итоге они могут попасть в различающиеся состояния так, как это показано на рис.19.18.

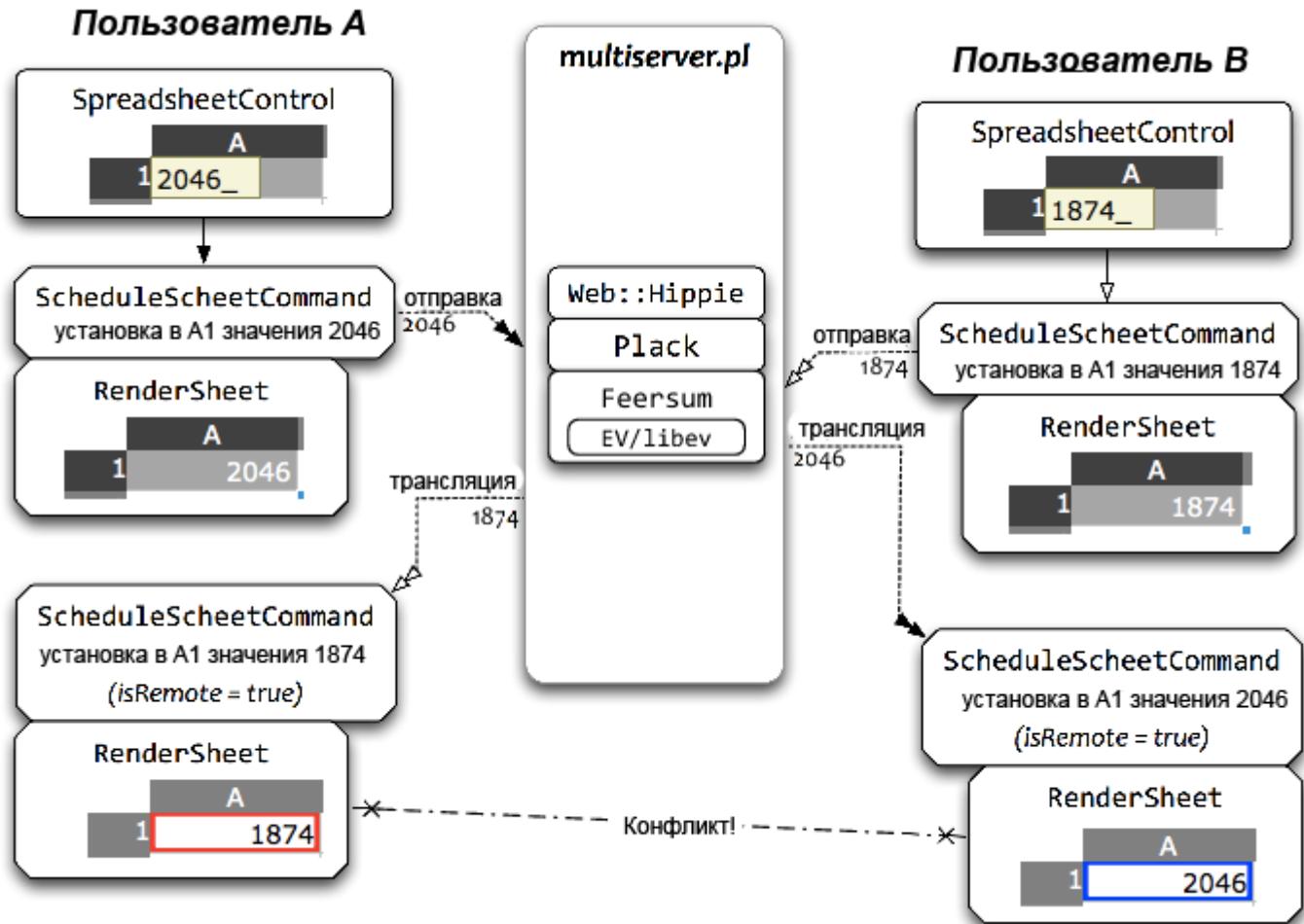


Рис.19.18: Конфликт, связанный с состоянием гонки (Race Condition)

Мы можем решить это с помощью встроенного механизма undo/redo, имеющегося в SocialCalc, так, как показано на рис.19.19.

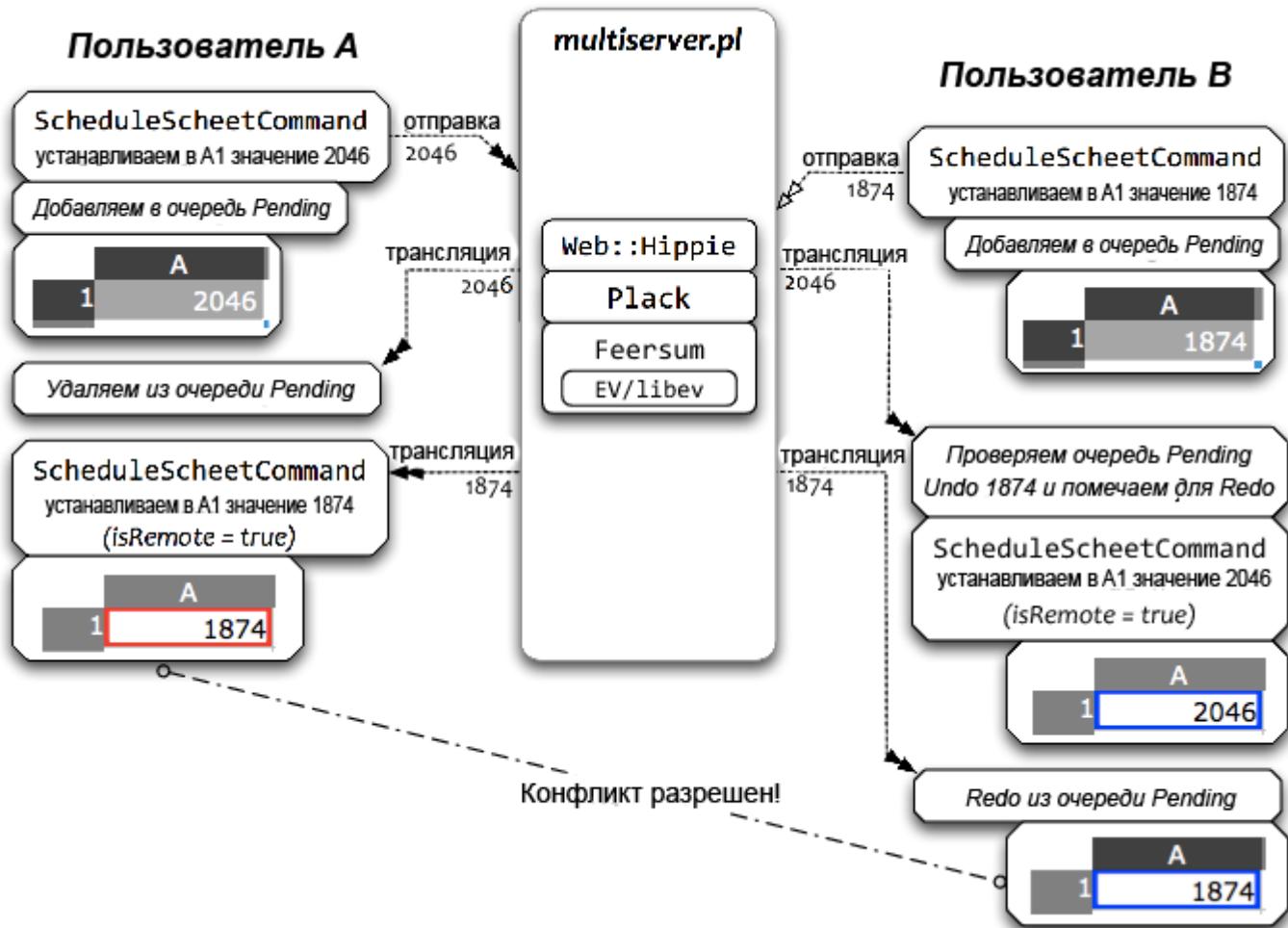


Рис.19.19: Разрешение конфликта, связанного с состоянием гонки (Race Condition)

Процесс, используемый для разрешения конфликта заключается в следующем. Когда клиент рассыпает команду, он добавляет команду в очередь ожидания команд Pending. Когда клиент получает команду, он проверяет, не противоречит ли полученная команда той команде, которая находится в очереди ожидания команд Pending.

Если очередь отложенных команд Pending пуста, то команда просто выполняется как дистанционно исполняемое действие. Если дистанционная команда находит в очереди Pending совпадающую команду, то локальный команда удаляется из очереди.

В противном случае, клиент проверяет, есть ли в очереди команды, которые противоречат принятой команде. Если есть противоречие команды, клиент сначала отменяет эти команды (операция Undo) и помечает их для повторного выполнения их впоследствии (операция Redo). После отмены действий конфликтующих команд (если таковые имеются), дистанционная команда выполняется обычным образом.

Когда от сервера поступает команда, отмеченная для повторного выполнения (операция Redo), клиент выполнит ее снова, а затем удалит ее из очереди.

19.7.3. Дистанционное управление курсором

Даже когда проблема с состоянием гонки решена, все еще есть шанс, что кто-нибудь случайно изменит содержимое ячейки, которое в настоящее время редактирует другой пользователь. Простым решением является рассылка каждым пользователем позиции своего курсора всем остальным пользователям с тем, чтобы все видели, в какой ячейке выполняется работа.

Чтобы реализовать эту идею, мы добавляем к событию MoveECellCallback еще один обработчик broadcast:

```
editor.MoveECellCallback.broadcast = function(e) {
    hpipe.send({
        type: 'ecell',
        data: e.ecell.coord
    });
};

$(hpipe).bind("message.ecell", function (e, d) {
    var cr = SocialCalc.coordToCr(d.data);
    var cell = SocialCalc.GetEditorCellElement(editor, cr.row, cr.col);
    // ... оформляем ячейку в стиле, соответствующему ситуации, когда ней работает другой пользователь ...
});
```

Чтобы пометить ячейку, которая получила фокус в таблицах, обычно используют цветные границы. Однако для ячейки может быть определено ее собственное свойство border (граница), а так как границы окрашены одинаково, в одной о той же ячейке может находиться только один курсор.

Поэтому в браузерах, в которых поддерживается CSS3, мы для представления нескольких курсоров в одной и той же ячейке используем свойство box-shadow:

```
/* Два курсора в в одной и той же ячейке */
box-shadow: inset 0 0 0 4px red, inset 0 0 0 2px green;
```

На рис.19.20 показано, как будет выглядеть экран, когда с одной и той же таблицей работают четыре человека.

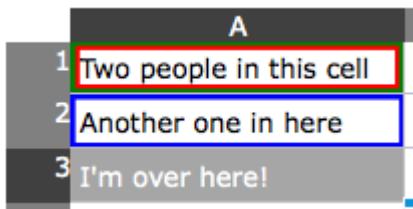


Рис.19.20: Четыре пользователя редактируют одну и ту же таблицу

19.8. Усвоенные уроки

Мы выпустили SocialCalc 1.0 19 октября 2009 года в 30-летнюю годовщину первого выпуска VisiCalc. Опыт сотрудничества с коллегами в Socialtext под руководством Дэна Бриклина был очень ценен для меня, и я хотел бы поделиться некоторыми уроками, которые я за это время усвоил.

19.8.1. Главный конструктор с ясным видением

В работе [Bro10] Фред Брукс (Fred Brooks) утверждает, что если мы ориентируемся на стройную концепцию архитектуры, обсуждения при создании сложных систем должны быть более прямыми, а не выводятся из других решений. По словам Брукса, формулировку такую единой архитектурно целостную концепцию лучше всего отдать на откуп одному человеку:

Поскольку концептуальная целостность является самым важным атрибутом отличной архитектуры и т.к. это должно быть результатом размышлений одного человека или нескольких, работающих как *единое целое*, мудрый менеджер смело доверит всю задачу создания архитектуры одаренному главному конструктору.

В случае с SocialCalc наличие Трейси Рагглса (Tracy Ruggles) в качестве нашего главного конструктора, имеющего пользовательский опыт, было для проекта ключевым фактором, позволяющим свести все к общему видению. Поскольку движок, лежащий в основе SocialCalc, был настолько податлив, был очень велик соблазн создать массу мелких функций. Способность Трейси общаться с помощью проектных эскизов действительно помогли нам представить все функции так, как их интуитивно чувствуют пользователи.

19.8.2. Wiki для сообщества, работающего с проектом

Прежде, чем я присоединился к проекту SocialCalc, проект уже был в работе в течение более двух лет, но я смог его догнать и менее чем за неделю начал вносить в него свой вклад, причем просто благодаря тому, что все было в *wiki*. Весь процесс был запечатлен в *wiki*-страницах и таблицах SocialCalc, начиная с первых аккордов разработки архитектуры и до самой последней матрицы поддержки браузеров.

Чтение рабочего проекта быстро переводило меня на ту же самую страницу, что и других, причем без обычного размахивания руками над головой, обычно сопровождаемого вхождение в проект нового члена команды.

Это было бы невозможно в традиционных проектах с открытым кодом, когда большая часть обсуждений происходит по IRC и по спискам рассылок, а *wiki*-технология (если таковая имеется) используется только для документирования, а также для ссылок на ресурсы, используемые при разработке. Новичку гораздо труднее восстановить контекст проекта по неструктурированным журналам IRC и почтовым архивам.

19.8.3. Объединяя разные часовые пояса

Когда Давид Хейнемайер Ханссон (David Heinemeier Hansson), создатель Ruby on Rails, первый раз попал в команду 37signals, он как-то заметил о пользе распределенных команд: «Семь часовых поясов между Копенгагеном и Чикаго фактически означали, что мы могли делать очень много с небольшими перерывами». Это также оказалось верным, когда при разработке SocialCalc Тайбэй и Пало-Альто были разделены девятью часовыми поясами.

Мы часто завершали весь цикл от запроса до реализации в течение 24-часового рабочего дня, причем каждый вопрос решался дня одним человеком в течение 8-часового рабочего днем по его местному времени. Такой асинхронный стиль сотрудничества вынуждал нас создавать самодокументированные артефакты (эскизы проекта, код и тесты), что, в свою очередь, значительно улучшило наше доверие друг к другу.

19.8.4. Оптимизация для удовольствия

В 2006 году в своем докладе для конференции CONISLI [Tan06], я изложил в форме нескольких заметок свой собственный опыт руководства распределенной командой, реализующей язык Perl 6.. Среди них - «всегда иметь план», «снисходительность ведет к вседозволенности», «избегать тупиков», «искать идеи, а не консенсус» и «предлагать эскиз идеи вместе с кодом», которые особенно актуальны для небольших распределенных команд.

При разработке SocialCalc, мы особое внимание уделяли тому, как знания, касающиеся проекта, распределялись среди членов команды, совместно работающих над кодом, поэтому никто из нас не мог стать узким местом в проекте.

Кроме того, мы заранее решали споры по помощи кодирования различных вариантов с тем, чтобы изучить какую-нибудь часть проекта, и, когда появлялись лучшие проектные решения, мы не боялись заменять ими полностью рабочие прототипы.

Эти нормы поведения помогли нам воспитать чувство предвидения и взаимопомощи, и, несмотря на отсутствие непосредственного взаимодействия друг с другом, свести политические вопросы к минимуму и сделать так, чтобы работа над проектом SocialCalc доставляла удовольствие.

19.8.5. Управление разработкой с использованием «описаний - проверок»

До прихода в Socialtext, как можно видеть по спецификациям Perl 6 [7], в которых мы добавляли официальные наборы тестов к спецификациям языка, я выступал за подход «чередования тестов со спецификациями». Впрочем, еще были Кен Пьер (Ken Pier) и Мэтт Хюссер, Heusser, команда аналитиков SocialCalc, которая действительно открыла мне глаза на то, как можно перейти на следующий уровень, преобразовав тесты в *исполняемые спецификации*.

В главе 16 в [GR09] Мэтт разъяснил наш процесс разработки с использованием подхода «описание - проверка» следующим образом:

Основной единицей работы является «описание», которое является документом с чрезвычайно простыми спецификациями. В нем находится краткое описание предлагаемой функции и даются примеры того, что должно произойти, чтобы считать эту функцию реализованной; мы называем эти примеры «приемо-сдаточными проверками» и описываем их на простом английском языке.

В начале «описания» владелец продукта делает добросовестную первую попытку создать приемо-сдаточную проверку, которая будет расширена разработчиками и тестерами перед тем, как кто-нибудь из разработчиков напишет какую-нибудь сточку кода.

Такие «описания - проверки» затем преобразовываются в wiki-тесты, табличный язык спецификаций, предложенный Уордом Каннингемом во фреймворке FIT [8] и позволяющий управлять фреймворками автоматического тестирования, например, `Test::WWW::Mechanize` [9] и `Test::WWW::Selenium` [10].

Трудно переоценить пользу использования подхода «описание - проверки» в качестве обобщенного языка для формулировки и проверки требований. Он сыграл важную роль в уменьшении массы недоразумений, и позволил в наших ежемесячных выпусках практически полностью избежать возврата к старым проблемам.

19.8.6. Открытый исходный код с лицензией CPAL

Последним, но не менее важным уроком, который интересен сам по себе, является модель открытого исходного кода, выбранную нами для SocialCalc.

В Socialtext для SocialCalc была создана лицензия Common Public Attribution License [11]. Лицензия CPAL, разработанная на основе лицензии Mozilla Public License, предназначена для того, чтобы позволить автору исходного текста требовать от пользователя исходного текста отображать информацию о себе в пользовательском интерфейсе производных программ, а также имеет пункт, касающийся использования в сети, в котором требуется в производной программе в случае, если она используется в сети в качестве сервиса, указывать те же самые условия распространения, что и в исходной лицензии.

После одобрения лицензии в Open Source Initiative [12] и в Free Software Foundation [13], мы обнаружили, что такие известные сайты, как Facebook [14] и Reddit [15], решили выпустить исходный код своей платформы под лицензией CPAL, что очень обнадеживает.

Поскольку лицензия CPAL является лицензией «weak copyleft» (т. е. с весьма слабыми ограничениями — прм.пер.), разработчики могут свободно комбинировать ее с либо бесплатным, либо с проприетарным программным обеспечением, и единственное, что нужно, это создать собственную

модификацию SocialCalc. Это позволило различным сообществам адаптировать SocialCalc и сделали его еще более мощным.

Движок для работы с электронными таблицами, имеющий открытый исходный код, предоставляет много интересных возможностей и если вы сможете найти способ использовать SocialCalc в вашем любимом проекте, мы, определенно, хотели бы об этом узнать.

Примечания

1. <https://github.com/audreyt/wikiwyg-js>
2. <http://one.laptop.org/>
3. http://seeta.in/wiki/index.php?title=Collaboration_in_SocialCalc
4. <http://search.cpan.org/dist/Web-Hippie/>
5. <http://about.digg.com/blog/duistream-and-mxhr>
6. <https://github.com/gimite/web-socket-js>
7. <http://perlcabal.org/syn/S02.html>
8. <http://fit.c2.com/>
9. <http://search.cpan.org/dist/Test-WWW-Mechanize/>
10. <http://search.cpan.org/dist/Test-WWW-Selenium>
11. <https://www.socialtext.net/open/?cpal>
12. <http://opensource.org/>
13. <http://www.fsf.org>
14. <https://github.com/facebook/platform>
15. <https://github.com/reddit/reddit>

20. Фреймворк Telepathy

Telepathy является модульным фреймворком для коммуникаций в режиме реального времени, в котором можно обрабатывать голосовые, текстовые, видео сообщения, осуществлять передачу файлов и так далее. Фреймворк Telepathy уникален не столько тем, что он абстрагирует особенности различных протоколов передачи мгновенных сообщений, а тем, что он воплощает идею коммуникации как сервиса, во многом похожего на то, как печать является сервисом, доступным одновременно для многих приложений. Для достижения этого в Telepathy интенсивно используется шина передачи сообщений D-Bus и модульная архитектура.

Коммуникации, как сервис, невероятно полезны, поскольку позволяют нам с помощью них выйти за рамки отдельного приложения. Нам предлагается много интересных вариантов применения: можно видеть наличие контакта в почтовом приложении, начать с ним общение, использовать этот контакт для передачи ему файла прямо из браузера файлов или, используя контакты, выполнять совместную работу внутри приложений — возможность, известная в Telepathy как *Tubes туннели*.

Фреймворк Telepathy был создана Робертом Маккуином (Robert McQueen) в 2005 году, и с того времени разрабатывался и поддерживался несколькими компаниями и отдельными разработчиками, в том числе компанией Collabora, одним из сооснователей которой является Маккуин.

Шина передачи сообщений D-Bus

Шина D-Bus является шиной асинхронной передачи сообщений межпроцессного взаимодействия, что представляет собой остов большинства систем GNU/Linux, в том числе среди рабочего стола для GNOME и KDE. D-Bus является, прежде всего, архитектурой с общейшиной: приложения подключаются к шине (идентифицируемой адресом сокета) и могут либо передавать сообщения, адресованные другому приложению, подключенному к шине, либо осуществлять широковещательную передачу сигнала для всех, кто подключен к шине. Приложения, подключенные к шине, имеют шинный адрес, похожий на IP-адрес, и могут объявлять об использовании нескольких заре-

гистрированных за ними имен, похожих на имена DNS, например `org.freedesktop.Telepathy.AccountManager`. Все процессы взаимодействуют через демон D-Bus, с помощью которого происходит передача сообщений и регистрация имен.

С точки зрения пользователя, в каждой системе есть две шины. Системная шина представляет собой шиной, позволяющей пользователю взаимодействовать с компонентами, имеющимися в системе (принтерами, устройствами bluetooth, средствами управления аппаратурой и так далее), и которая совместно используется всеми пользователями системы. Сессионная шина является уникальной для каждого пользователя, то есть для каждого пользователя, который вошел в систему, имеется своя собственная сессионная шина, применяющаяся в пользовательских приложениях для связи их друг с другом. Когда через шину нужно передать большой объем трафика, в приложениях также можно создать собственную шину приложения, или можно создать шину типа «точка-точка» (peer-to-peer), управление которой не будет происходить при помощи демона `dbus-daemon`.

С помощью нескольких библиотек реализован протокол D-Bus, через который можно взаимодействовать с демоном D-Bus, например, помошью библиотек `libdbus`, `GDBus`, `QtDBus` и `python-dbus`. На эти библиотеки возложена обязанность отправки и получения сообщений D-Bus, преобразование типов из системы типов данных языков программирования в формат типов шины D-Bus, и публикация объектов в среде шины. Обычно в библиотеках также предоставляются удобные интерфейсы API, используемые для получения списков подключенных приложений и приложений, которые могут быть активированы, а также интерфесы для запроса имен, зарегистрированных вшине. На уровне шины D-Bus, всё это сделано с помощью вызовов методов опубликованного объекта, что выполняется демоном `dbus-daemon` самостоятельно.

Более подробную информацию о шине D-Busсмотрите по ссылке <http://www.freedesktop.org/wiki/Software/dbus>.

20.1. Компоненты фреймворка Telepathy

Telepathy является модульным фреймворком, в котором каждый модуль взаимодействует с другими через шину сообщений D-Bus. Причем, чаще всего через сессионную шину пользователя. Это взаимодействие подробно описано в спецификациях фреймворка Telepathy [2]. На рис.20.1 показаны следующие компоненты Telepathy:

- Менеджер соединений (Connection Manager) предоставляет интерфейс между Telepathy и отдельными коммуникационными сервисами. Например, есть менеджер соединений для XMPP, для SIP, для IRC, и т.д. Добавление поддержки нового протокола в Telepathy представляет собой написание нового менеджера соединений.
- Сервис менеджера аккаунтов (Account Manager) отвечает за хранение коммуникационных аккаунтов пользователей и за установку, когда будет получен запрос, подключения к каждому аккаунту через соответствующий менеджер соединений.
- Роль диспетчера каналов (Channel Dispatcher) состоит в прослушивании во входящих каналах сигналов, передаваемых каждым менеджером соединений, и перенаправлении их к клиентам, которые указали, что они могут обрабатывать данный тип канала, например текстовые сообщения, голосовые и видео сообщения, выполнять передачу файлов, использовать туннели.
- Клиентские программы или клиенты фреймворка Telepathy обрабатывают или наблюдают за коммуникационными каналами. К их числу относятся как пользовательские интерфейсы, например, клиенты IM и VoIP, так и сервисы, например, такой, который регистрирует сообщения чата. Клиентские программы самостоятельно регистрируются в диспетчера канала, указывая список типов каналов, которые они хотели бы обрабатывать или за которыми они хотели наблюдать.

В текущей реализации фреймворка Telepathy, менеджер аккаунтов и диспетчер каналов представлены в виде единого процесса, который называется центром управления (Mission Control).

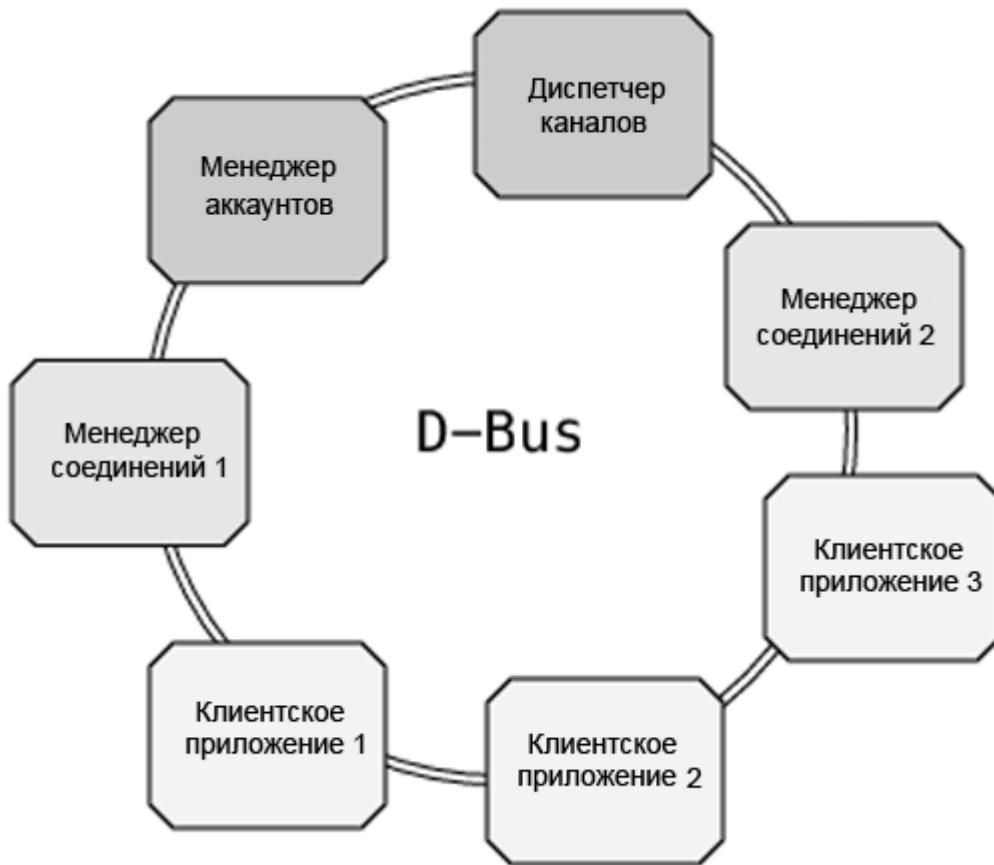


Рис.20.1: Пример компонентов фреймворка Telepathy

Такая модульная архитектура основывается на философии Дага Макилроя (Doug McIlroy): «Пишите программы, которые делают что-то одно, и делают это хорошо», и имеет несколько важных преимуществ:

- *Устойчивость к ошибкам (Robustness)*: Ошибка в одном из компонентов не должна приводить к краху всего сервиса.
- *Легкость разработки (Ease of development)*: Компоненты в работающей системе могут меняться, что не должно влиять на другие компоненты. Должна быть возможность проверять версию разрабатываемого модуля и сравнивать ее с версиями других модулей, которые работают надлежащим образом.
- *Независимость от языка программирования (Language independence)*: Компоненты могут быть написаны на любом языке программирования, для которого есть привязка (binding) к шине D-Bus. Если данный протокол лучше всего реализован на каком-то конкретном языке, вы можете написать на этом языке свой собственный менеджер соединений и он будет доступен для всех клиентов фреймворка Telepathy. Точно также, если вы решили разработать свой собственный пользовательский интерфейс на каком-то конкретном языке, то у вас имеется доступ ко всем имеющимся протоколам.
- *Лицензионная независимость (License independence)*: Компоненты могут использовать различные лицензии, которые были бы несовместимыми, если бы всё работало в одном процессе.
- *Независимость от интерфейсов (Interface independence)*: Поверх одних и тех же компонентов Telepathy можно разрабатывать большое количество пользовательских интерфейсов. Это позволяет пользоваться нативными интерфейсами в среде рабочего стола и в аппаратно реализуемых устройствах (например, GNOME, KDE, Meego, Sugar).
- *Безопасность (Security)*: Компоненты работают в различных адресных пространствах с весьма ограниченными правами. Например, типичному менеджеру соединений нужен всего

лишь доступ к сети и к сессионной шине D-Bus, что позволяет использовать нечто, похожее на SELinux, для ограничения доступа к компонентам.

Менеджер соединений управляет некоторым количеством соединений, каждое соединение представляет собой логическое подключение к некоторому коммуникационному сервису. В каждом аккаунте есть одно соединение. В соединении может быть много каналов. Каналы являются механизмом, через который происходит коммуникация. С помощью каналов реализовывают обмен мгновенными сообщениями IM, голосовые или видео вызовы, пересылку файлов, или какой-либо другие операции, требующие сохранение состояния (stateful operation). Соединения и каналы более подробно рассматриваются в разделе 20.3.

20.2. Как в Telepathy используется шина D-Bus

Взаимодействие компонентов Telepathy осуществляется через шину обмена сообщениями D-Bus, которая обычно является сессионной шиной пользователя. В D-Bus предоставляются возможности, обычные для систем межпроцессных взаимодействий: каждый сервис публикует объекты, имеющие строго определенные пути в пространстве имен, ведущие к этим объектам, например `/org/freedesktop/Telepathy/AccountManager` [3]. В каждом объекте реализовано несколько интерфейсов. Интерфейсы снова строго определены в пространстве имен, и имеют вид `org.freedesktop.DBus.Properties` и `ofdT.Connection`. В каждом интерфейсе предоставляются методы, сигналы и свойства, которые вы можете вызывать, слушать или опрашивать.

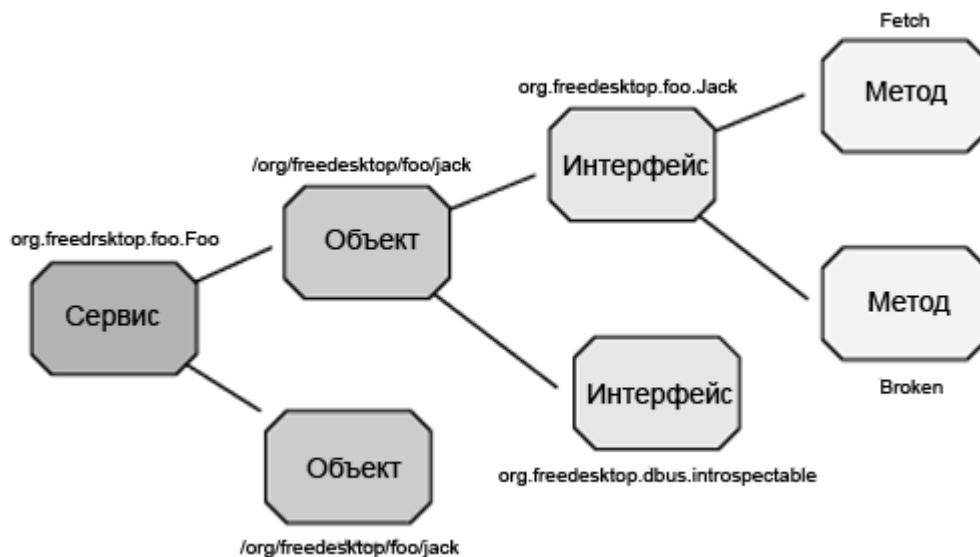


Рис.20.2: Концептуальное представление объектов, публикуемых сервисом D-Bus

Публикация объектов D-Bus

Публикация объектов D-Bus полностью выполняется библиотекой D-Bus, которая используется в данный момент. По сути, это отображение пути к объекту D-Bus в объект, используемый в программе и реализующий данные интерфейсы. Пути к объектам, публикуемые сервисом, предоставляются для доступа с помощью дополнительного интерфейса `org.freedesktop.DBus.Introspectable`.

Когда сервис получает входящий вызов метода, в котором указан путь назначения (например, `/ofdT/AccountManager`), на библиотеку D-Bus возлагается обязанность найти в программе объект, представляемый данным объектом D-Bus, а затем выполнить вызов соответствующего метода для этого объекта.

Интерфейсы, методы, сигналы и свойства, предоставляемые Telepathy, подробно описываются с помощью языка D-Bus IDL (Interface Definition Language, язык описания интерфейсов), в основе которого лежит язык XML, расширенный таким образом, чтобы можно было включать больше информации. Спецификации в таком виде можно анализировать автоматически и создавать документацию и привязки к различным языкам программирования.

Сервисы Telepathy публикуют различные объекты на шине. Центр управления публикует объекты для менеджера соединений и диспетчера каналов так, чтобы был доступ к их сервисам. Клиентские программы публикуют объект Client так, чтобы он был доступен для диспетчера каналов. Наконец, менеджеры соединений публикуют следующие объекты: объект сервиса, который может использоваться менеджером аккаунтов для запроса новых соединений, по одному объекту для каждого открытого соединения и по одному объекту для каждого открытого канала.

Хотя объекты D-Bus не имеют типов, а только интерфейсы, в Telepathy типы эмулируются несколькими способами. Путь к объекту, сообщающий нам, где находится объект, является соединением, каналом, клиентской программой, и т.д., хотя обычно вы уже знаете об этом, когда запрашиваете прокси доступ для этого объекта. Каждый объект реализует базовый интерфейс для его собственного типа, например `ofdT.Connection` или `ofdT.Channel`. Для каналов, это очень похоже на абстрактный базовый класс. Объекты каналов также имеют конкретный класс, определяемый типом данного канала. Снова это представлено с помощью интерфейса шины D-Bus. О типе канала можно узнать, прочитав свойство `ChannelType` в интерфейсе `Channel`.

Наконец, в каждом объекте реализуется ряд необязательных интерфейсов (как ни странно, тоже представляемые как интерфейсы шины D-Bus), которые зависят от возможностей протокола и менеджера соединений. Список имеющихся интерфейсов для данного объекта представлены в свойстве `Interfaces` базового класса объекта.

Для объектов соединений `Connection` типа `\code{ofdT.Connection}`, необязательные интерфейсы имеют имена вида - `ofdT.Connection.Interface.Avatars` (если в протоколе есть понятие аватаров), `ofdT.Connection.Interface.ContactList` (если в протоколе предоставлен реестр контактов – это делается не везде) и `ofdT.Connection.Interface.Location` (если в протоколе предоставлена геолокационная информация). Для объектов каналов `Channel` типа `ofdT.Channel`, конкретные классы имеют имена интерфейсов вида `ofdT.Channel.Type.Text`, `ofdT.Channel.Type.Call` и `ofdT.Channel.Type.FileTransfer`. Необязательный интерфейс точно также, как и объекты `Connection`, имеют имена вида `ofdT.Channel.Interface.Messages` (если канал может передавать и получать текстовые сообщения) и `ofdT.Channel.Interface.Group` (если данный канал подключен к группе, содержащей много контактов, например, к многопользовательскому чату). Так, например, в текстовом канале реализуются, как минимум, интерфейсы `ofdT.Channel`, `ofdT.Channel.Type.Text` и `Channel.Interface.Messages`. Если это многопользовательский чат, в нем также будет реализован интерфейс `ofdT.Channel.Interface.Group`.

Почему свойство `Interfaces`, а не анализ в самойшине D-Bus?

Вы можете поинтересоваться, почему в каждом базовом классе реализовано свойство `Interfaces`, вместо того, чтобы воспользоваться средствами анализа, имеющимися в самойшине D-Bus, которые бы нам сообщали, какие имеются интерфейсы. Дело в том, что в различных объектах каналов и соединений могут, в зависимости от возможностей канала или соединения, предлагаться различные интерфейсы, которые отличаются друг от друга, но в большинстве реализаций средств анализа D-Bus предполагается, что все объекты одного и того же класса объектов будут иметь одинаковые интерфейсы. Например, в `telepathy-glib` интерфейсы D-Bus, перечисляемые с помощью средств анализа D-Bus, берутся из реализаций классов интерфейсов объекта, которые определяются статически на этапе компиляции. Мы справляемся со всем этим благодаря наличию средств анализа вшине D-Bus, предоставляющих данные обо всех интерфейсах, которые могли бы быть в

объекте, и благодаря использованию свойства `Interfaces`, указывающего, какие интерфейсы действуют на самом деле.

Хотя в самой шине D-Bus нет проверки того, что в объектах соединений присутствуют интерфейсы, относящиеся только к подключениям, и т. д. (поскольку в шине D-Bus не используется концепция типов, а лишь концепция интерфейсов, именуемых произвольным образом), мы для того, чтобы реализовывать такую проверку внутри языковых привязок фреймворка Telepathy, можем пользоваться информацией, имеющейся в спецификациях Telepathy.

Почему и как был расширен язык спецификаций IDL

В существующем языке спецификаций IDL шины D-Bus определяются имена, аргументы, ограничения доступа и сигнатуры методов, свойств и сигналов для типов D-Bus. Но в нем не поддерживается документирование, механизмы привязок и именованные типы.

Чтобы обойти эти ограничения, в XML было добавлено новое пространство имен XML, предлагающее требуемую информацию. Это пространство имен для того, чтобы его можно было использовать другими интерфейсами API шины D-Bus, было разработано в наиболее обобщенном виде. Были добавлены новые элементы, среди которых есть возможность прямо внутрь интерфейса API добавлять документацию, описание обоснования тех или иных решений, вводную часть, отмечать устаревшие версии и возможные исключения, возникающие в методах.

Сигнатуры типов D-Bus являются низкоуровневым описанием, определяющим какие данные должны быть сериализованы использования в шине. Сигнтура типов D-Bus могут выглядеть, например, как `(ii)` (что представляет собой структуру, содержащую два элемента `int32`), или может быть более сложной. Например, `a{sa(usuu)}` - это ассоциативный массив, который ассоциирует строку с массивом структур, содержащих элементы `uint32`, `string`, `uint32`, `uint32` (смотрите рис.20.3). Эти типы, хотя и описывают формат данных, не придают никакого семантического смысла информации, хранящейся в объектах этих типов.

Чтобы предоставить программистам семантическую прозрачность и усилить типизацию привязок к языкам программирования, были добавлены новые элементы, позволяющие именовать простые типы, структуры, ассоциативные массивы, перечислимые типы (`enums`) а также флаги, предоставившие их сигнатуры, а также документацию. Также были добавлены элементы, эмулирующие наследование объектов шины D-Bus.

Тип (ii)

Структура:
int32
int32

Тип a{sa(usuu)}

Отображение:

string —

Массив:

Структура:
uint32
uint32
string
uint32

Структура:

string —

Массив:

Рис.20.3: Тип (ii) и тип a{sa(usuu)} шины D-Bus

20.2.1. Хэндлы (Handles)

Хэндлы используются в Telepathy для представления идентификаторов (например, контактов или названия чат-комнат). Они являются беззнаковыми целочисленными значениями, назначаемые менеджером соединений так, что кортеж (соединение, тип хэндла, хэндл) уникальным образом определяет конкретный контакт или чат-комнату.

Из-за того, что различные протоколы по-разному обращаются с идентификаторами (например, это касается чувствительность к регистру, ресурсам), хэндлы предоставляют клиентам способ определить, являются ли два идентификатора одним и тем же одинаковыми. Можно запросить хэндлы различных идентификаторов, и если номера хэндов совпадают, то идентификаторы указывают на один и тот же контакт или чат-комнаты.

Правила нормализации идентификаторов для различных протоколов различны, поэтому было бы ошибкой, если бы клиентские программы при их сравнении сравнивали бы их как строки. Например, escher@tuxedo.cat/bed и escher@tuxedo.cat/litterbox являются двумя экземплярами одного и того же контакта (escher@tuxedo.cat) в протоколе XMPP и, следовательно, они имеют одинаковый хэндл. Клиентские программы могут запрашивать каналы либо по идентификатору, либо по хэндлу, но при сравнении они всегда должны использовать хэндлы.

20.2.2. Обнаружение сервисов Telepathy

Некоторые сервисы, такие как менеджер аккаунтов и диспетчер каналов, которые существуют всегда, имеют общезвестные имена, которые определены в спецификации Telepathy. Однако, имена менеджеров соединений и клиентских программ строго не определяются и их нужно обнаруживать динамически.

В Telepathy нет сервиса, который бы отвечал за регистрацию запущенных менеджеров соединений и клиентских программ. Вместо этого, те, кому это интересно, слушают шину D-Bus с тем, чтобы

узнавать о появлении нового сервиса. Демон шины D-Bus посыпает сигнал каждый раз, когда нашине появляется новый именованный сервис D-Bus. Имена клиентов и менеджеров соединений начинаются с известных префиксов, определяемых спецификацией, и в именах новых сервисов будут присутствовать эти префиксы.

Преимущество такого подхода в том, что для него абсолютно не нужно запоминать состояния (stateless). Когда запускается компонент Telepathy, он может спросить у демона шины (у которого есть канонический список, составленный на основе открытых соединений) о том, какие сервисы в настоящий момент работают. Например, если происходит сбой в работе менеджера аккаунтов, то он может посмотреть список запущенных соединений и заново создать ассоциации между соединениями и объектами аккаунтов.

Соединения тоже являются сервисами

Как и сами менеджеры соединений, соединения точно также публикуются как сервисы шины D-Bus. Гипотетически менеджеру соединений разрешается выделять каждое соединение в виде отдельного процесса, но до настоящего времени это в менеджере соединений пока реализовано. На практике это позволяет находить все запущенные соединения, запросив у демона шины D-Bus все соединения, которые начинаются с префикса `ofdT.Connection`.

Диспетчер каналов также использует этот же метод для поиска клиентских программ Telepathy. Они начинаются с имени `ofdT.Client`, например, `ofdT.Client.Logger`.

20.2.3. Снижение трафика шины D-Bus

Первоначальные версии спецификации Telepathy были причиной излишнее большого трафика вшине D-Bus из-за того, что огромное количество клиентских программ, подключенных кшине, вызывали методы, запрашивающие необходимую им информацию. В более поздних версиях Telepathy эта проблема была решена с помощью ряда оптимизаций.

Вызовы отдельных методов были заменены свойствами шины D-Bus. В первоначальной спецификации для каждого свойства объекта использовался отдельный вызов метода: `GetInterfaces`, `\code{GetChannelType}`, и т.д. Запрос всех свойств объекта требовал вызовов нескольких методов, у каждого из которых имелись свои накладные расходы. Если пользоваться свойствами шины D-Bus, то все это можно получить за один раз с помощью стандартного метода `GetAll`.

Более того, целый ряд свойств канала остается неизменным в течение всего времени существования канала. К ним относятся такие свойства, как тип канала, те интерфейсы, через которые подключаются кканалу, а также инициатор открытия канала. Например, для канала передачи файлов среди них также будут указаны размер файла и тип содержимого файла.

Был добавлен новый сигнал, с помощью которого объявляется о создании каналов (как входящих, так и в ответ на исходящие запросы) и в котором находится хэш-таблица неизменяемых свойств. Она может быть передана напрямую в прокси конструктор канала (смотрите раздел 20.4), что позволяет заинтересованным в этой информации клиентским программам не запрашивать ее по отдельности.

Аватары пользователей передаются пошине в виде массивов байтов. Хотя в Telepathy уже используются токены для ссылок на аватары, что позволяет клиентским программам знать, что им нужен новый аватар, и не загружать ненужные аватары, каждая клиентская программа должна была отдельно запрашивать аватар через метод `RequestAvatar`, который в своем ответе возвращает аватар. Таким образом, когда менеджер соединений посыпал сигнал о том, что уконтакта обновился аватар, нужно было для получения аватара делать несколько отдельных запросов и из-за этого аватар передавался пошине сообщений несколько раз.

Эта проблема была решена с помощью добавления нового метода, который не возвращает аватар (он ничего не возвращает). А вместо этого он добавляет аватар в очередь запросов. Получение аватара по сети осуществляется по сигналу `AvatarRetrieved`, который могут слушать все заинтересованные клиентские программы. Это значит, что данные аватара требуется передавать по шине всего один раз, а затем они становятся доступными для всех клиентских программ, которые в нем заинтересованы. Поскольку запрос клиентской программы находился в очереди, все последующие запросы клиентских программ могли игнорироваться до тех пор, пока не будет отправлен сигнал о готовности аватара `AvatarRetrieved`.

Всякий раз, когда нужно загрузить большое количество контактов (например, при загрузке реестра контактов), нужно запрашивать большое количество информации: алиасы контактов, аватары, функциональные возможности, их принадлежность к группам, а также, возможно, их местоположение, адреса, номера телефонов. Ранее, в `Telepathy` для этого требовалось бы по одному вызову отдельного метода на каждую группу информации (большинство вызовов API, например, `GetAliases` уже принимало списки контактов), что приводило более, чем к полудесятку или даже к большему числу вызовом методов.

Чтобы решить эту проблему, был добавлен интерфейс `Contacts`. Он позволяет с помощью вызова единственного метода возвращать информацию из нескольких интерфейсов. Чтобы можно было добавлять атрибуты контактов (`Contact Attributes`), была расширена спецификация `Telepathy`: для получения информации о контакте был использован метод `GetContactAttributes`, который возвращал свойства, указываемые в виде полных имен из пространства имен, и который заменил вызовы других методов. Клиентская программа вызывает метод `GetContactAttributes` со списком контактов и интерфейсов, которые ее интересуют, а обратно она получает карту контактов, в которой для всех атрибутов контактов указываются их значения.

Небольшое количество кода сделает это более понятным. Запрос выглядит следующим образом:

```
connection[CONNECTION_INTERFACE_CONTACTS].GetContactAttributes(
    [ 1, 2, 3 ], # contact handles
    [ "ofdT.Connection.Interface.Aliasing",
      "ofdT.Connection.Interface.Avatars",
      "ofdT.Connection.Interface.ContactGroups",
      "ofdT.Connection.Interface.Location"
    ],
    False # don't hold a reference to these contacts
)
```

а ответ может выглядеть следующим образом:

```
{ 1: { 'ofdT.Connection.Interface.Aliasing/alias': 'Harvey Cat',
      'ofdT.Connection.Interface.Avatars/token': hex string,
      'ofdT.Connection.Interface.Location/location': location,
      'ofdT.Connection.Interface.ContactGroups/groups': [ 'Squid House' ],
      'ofdT.Connection/contact-id': 'harvey@nom.cat'
    },
  2: { 'ofdT.Connection.Interface.Aliasing/alias': 'Escher Cat',
      'ofdT.Connection.Interface.Avatars/token': hex string,
      'ofdT.Connection.Interface.Location/location': location,
      'ofdT.Connection.Interface.ContactGroups/groups': [],
      'ofdT.Connection/contact-id': 'escher@tuxedo.cat'
    },
  3: { 'ofdT.Connection.Interface.Aliasing/alias': 'Cami Cat',
      :     :     :
    }
}
```

20.3. Соединения, каналы и клиентские приложения

20.3.1. Соединения

Соединение создается менеджером соединений с тем, чтобы подключиться к отдельному протоколу/аккаунту. Например, подключение к аккаунтам XMPP `escher@tuxedo.cat` и `cami@egg.cat` должно в результате привести к созданию двух соединений, каждое из которых представлено объектом на шине D-Bus. Соединения обычно настраиваются менеджером аккаунтов для аккаунтов, имеющихся в данный момент.

В соединениях предоставляются некоторые обязательные функциональные возможности, необходимые для того, чтобы управлять и следить за состоянием сетевых соединений, а также для того, чтобы выполнять запрос каналов. В них также может быть предоставлен, в зависимости от возможностей протокола, ряд дополнительных возможностей. Они представляются как дополнительные интерфейсы шины D-Bus (так, как это рассказывалось в предыдущем разделе), которые перечисляются в свойстве `Interfaces` соединения.

Обычно соединениями управляет менеджер аккаунтов, созданный с использованием свойств соответствующих аккаунтов. Менеджер аккаунтов также будет для каждого аккаунта синхронизировать присутствие пользователя в соответствующем соединении и может для указанного аккаунта запросить путь к соединению.

20.3.2. Каналы

Каналы являются механизмом, через который осуществляются коммуникации. Обычно каналом может быть обмен мгновенными сообщениями IM, голосовой или видео звонок или передача файла, но каналы также можно использовать для предоставления состояний самого сервера в случае, когда требуется сохранять состояние (например, поиск чат-комнат или контактов). Каждый канал представлен объектом шины D-Bus.

Каналы обычно образуются между двумя или большим количеством пользователей, одним из которых являетесь вы сами. Обычно у канала есть целевой идентификатор, который является либо еще одним контактом в случае соединения типа «один-один», либо идентификатором чат-комнаты в случае соединения сразу со многими пользователями (например, с чат-комнатой). В многопользовательских каналах предоставляется интерфейс `Group`, который позволяет вам отслеживать, какие контакты в данный момент подключены к каналу.

Каналы принадлежат соединению, и их запрос происходит из менеджера соединений обычно через диспетчер каналов; либо они создаются самим соединением в ответ на событие сети (например, входящий чат), а затем передаются диспетчеру каналов для диспетчеризации.

Тип канала определяется свойством канала `ChannelType`. Основные возможности, методы, свойства и сигналы, которые необходимы для данного типа каналов (например, для отправки и приема текстовых сообщений), определены в соответствующем интерфейсе `Channel.Type` шины D-Bus, например, в `Channel.Type.Text`. В некоторых типах каналов могут быть реализованы дополнительные возможности (например, шифрование), которые представлены в качестве дополнительных интерфейсов, перечисляемых в свойстве `Interfaces`. Например, текстовый канал, который подключается к пользователю многопользовательской чат-комнаты, может иметь интерфейсы, перечисленные в таблице 20.1.

Таблица 20.1: Пример канала текстовых сообщений

Свойство	Назначение
<code>odft.Channel</code>	Общие возможности для всех каналов
<code>odft.Channel.Type.Text</code>	Тип канала, включающий в себя возможности, общие для всех каналов обмена текстовыми со

	общениями
odfT.Channel.Interface.Messages	Средства обмена сообщениями, имеющими расширенные возможности форматирования
odfT.Channel.Interface.Group	Компоненты перечисления (list), отслеживания (track), приглашения (invite) и одобрения (approve) для этого канала
odfT.Channel.Interface.Room	Чтение и назначение таких свойств, как тема чат-комнаты

Каналы списков контактов: ошибка

В первых версиях спецификации Telepathy списки контактов рассматривались как тип канала. Было несколько списков контактов, определяемых сервером (пользователи-подписчики, пользователи, публикующие сообщения, заблокированные пользователи), которые можно было запросить у каждого соединения. Затем точно также, как и в случае с многопользовательским чатом, можно было содержимое списка получить с помощью интерфейса Group.

Первоначально это позволяло создавать канал только после того, как список контактов был полностью получен, на что в некоторых протоколах затрачивается много времени. Клиент мог запрашивать канал когда угодно, и получать его, как только он был готов, но для пользователей с большим количеством контактов это означало, что запрос мог вызвать таймаут.

Группы контактов (например, Friends - Друзья) также предоставлялись в виде каналов, по одному каналу на каждую группу. Это при разработке клиентских приложений, которые работали с ними, приводило к исключительно трудным ситуациям. Для таких операций, как получение списка групп, к которым принадлежал контакт, требовалось в клиентском приложении писать код значительного размера. Более того, поскольку информация была доступна только через каналы, такие свойства, как группы контактов или состояние подписки, нельзя было публиковать через интерфейс Contacts.

Теперь оба этих вида каналов были заменены интерфейсами, предоставляемыми самим соединением, которые предоставляют информацию о реестре контактов в виде, более удобном для авторов клиентских приложений, и которое, включает в себя информацию о состоянии подписки контакта (тип enum), списке групп, в которые входит контакт, и списке контактов в группе. Когда список контактов будет подготовлен, об этом будет сообщено с помощью сигнала.

20.3.3. Запрос каналов, свойств каналов и диспетчеризация

Запрос каналов осуществляется с указанием карты свойств, которые, как вы желаете, должны предоставляться каналом. Обычно при запросе канала указывается тип канала, тип целевого хэндла (контакт или чат-комната) и сама цель. Однако, при запросе канала могут также указываться такие свойства, как имя или размер файла в случае передачи файлов, будет ли при вызовах сразу включаться аудио и видео, какие каналы из существующих следует объединить в вызов типа «конференция» или на каком сервере контактов осуществлять поиск контакта.

Свойства в запрашиваемом канале являются свойствами, определяемыми в спецификациях фреймворка Telepathy, например, свойство ChannelType (смотрите таблицу 20.2). Указывается полностью квалифицированное пространство имен интерфейса, откуда берутся свойства, которые можно указывать в запросах каналов и которые, согласно спецификациям фреймворка Telepathy, помечены как requestable (запрашиваемые).

Таблица 20.2: Примеры запросов каналов

Свойство	Значение
ofdT.Channel.ChannelType	ofdT.Channel.Type.Text
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat

В более сложном примере, приведенном в таблице 20.3, делается запрос канала передачи файлов. Обратите внимание на то, как запрашиваемые свойства квалифицируются с помощью интерфейсов, из которых они берутся. Для упрощения показаны не все необходимые свойства.

Таблица 20.3: Запрос канала передачи файлов

Свойство	Значение
ofdT.Channel.ChannelType	ofdT.Channel.Type.FileTransfer
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat
ofdT.Channel.Type.FileTransfer.Filename	meow.jpg
ofdT.Channel.Type.FileTransfer.ContentType	image/jpeg

Каналы могут быть в двух состояниях *created* (созданы) или *ensured* (гарантируется, что они будут созданы). Гарантирование создания канала означает, что он будет создан только том в случае, если он еще не существует. Запрос на создание канала приведет либо к тому, что будет создан абсолютно новый канал, либо запрос завершится ошибкой в случае, если несколько копий такого канала не могут существовать одновременно. Обычно вам потребуется лишь гарантировать, чтобы были созданы каналы передачи текстовых сообщений и каналы звонков (т.е. вам достаточно, чтобы для разговоров с одним человеком был открыт только один канал, и, более того, во многих протоколах не поддерживает одновременно несколько разговоров с одним и тем же самым контактом), а создавать каналы нужно будет только передачи файлов и в случае каналов, в котором сохраняется их собственное состояние (*stateful channels*).

Оповещение о только что созданных каналах (по запросу или как-нибудь иначе) осуществляется с помощью сигнала, поступаемого из соединения. В этом сигнале содержится карта неизменяемых свойств канала (*immutable properties*). Это такие свойства, для которых гарантированно, что они не будут изменяться в течение времени существования канала. К свойствам, которые рассматриваются как *immutable*, обычно относятся следующие: тип канала, тип целевого хэндла, цель канала, инициатор создания канала и список реализуемых интерфейсов. Ясно, что свойства, описывающие текущее состояние канала не являются неизменяемыми.

Запрос каналов — старый подход

Изначально, каналы можно было запрашивать по их типу, типу хэндла и цели. Этот подход был недостаточно гибким, т. к. не для всех каналов указывается цель их создания (например, каналы поиска контактов), а для некоторых запросов каналов требуется в первоначальный запрос включать дополнительную информацию (например, при передаче файлов, получении голосовой почты и отправке СМС).

Также было обнаружено, что при запросе канала хорошо было бы иметь два различных варианта поведения (либо всегда создавать гарантированно уникальный канал, либо просто обеспечивать, чтобы канал существовал), причем еще до того момента, когда соединении не будет принято решение о том, какому поведению следует отдать предпочтение. Поэтому старый метод был заменен новыми, более гибкими и явными.

Возврат неизменяемых свойств канала, когда вы его создаете или обеспечиваете, чтобы он был создан, позволяет гораздо быстрее создавать прокси-объект для канала. Теперь эту информацию

запрашивать не нужно. В ассоциативном массиве в таблице 20.4 показаны неизменяемые свойства, которые можно было бы добавить, когда делает запрос канала текстовых сообщений (например, в запросе канала, показанном в таблице 20.3). С целью упрощения некоторые свойства (включая TargetHandle и InitiatorHandle не показываются).

Таблица 20.4: Пример неизменяемых свойств, возвращаемых новым каналом

Свойство	Значение
ofdT.Channel.ChannelType	Channel.Type.Text
ofdT.Channel.Interfaces	{ [] } Channel.Interface.Messages, Channel.Interface.Destroyable, Channel.Interface.ChatState {} }
ofdT.Channel.TargetHandleType	Handle_Type_Contact (1)
ofdT.Channel.TargetID	escher@tuxedo.cat
ofdT.Channel.InitiatorID	danielle.madeley@collabora.co.uk
ofdT.Channel.Requested	True
ofdT.Channel.Interface.Messages.SupportedContentTypes	{ [] } text/html, text/plain {} }

Запрашивающая программа обычно отправляет запрос на канал диспетчеру каналов, передавая аккаунт, для которого производится запрос, запрос на канал и, возможно, имя требуемой программы-обработчика (это целесообразно, когда желательно, чтобы программа сама обрабатывала канал). Передача имени аккаунта вместо передачи соединения, означает, что диспетчер каналов может, если это потребуется, попросить менеджер аккаунтов перевести статус аккаунта в онлайн.

Когда запрос канала будет выполнен, диспетчер каналов либо передаст канал в указанный обработчик (handler), либо найдет подходящий обработчик (смотрите ниже обсуждение, относящееся к обработчикам и другим клиентским программам). Когда имя нужного обработчика является необязательным параметром, то у программы появляется возможность не интересоваться тем, ем, что в коммуникационных каналах при запросах каналов происходит за границами первоначального запроса, и отдавать все это на откуп тому обработчику, который для этого подходит наилучшим образом (например, запуская чат текстовых сообщений из вашего почтового клиента).



Рис.20.4: Запрос канала и диспетчеризация

Запрашивающая программа передает запрос канала в диспетчер каналов, который, в свою очередь, перенаправляет этот запрос в соответствующее соединение. Соединение выдает сигнал нового канала NewChannels, получаемый диспетчером каналов, который затем ищет подходящую клиентскую программу для обработки канала. Диспетчеризация входящих каналы, запросы на которые не делались, осуществляется аналогичным образом, когда сигнал, поступающий от соединения, получает диспетчер каналов, но, конечно, без первоначального запроса из программы.

20.3.4. Клиентские программы

Клиентские программы обрабатывают или наблюдают за входящими и исходящими коммуникационными каналами. Клиентской программой может быть все, что угодно, что зарегистрировано диспетчером каналов. Существует три вида клиентских приложений (хотя одно и то же клиентское приложение может, по желанию разработчика, относиться к двум или ко всем трем видам клиентских приложений):

- *Наблюдатели (Observers)*: Наблюдают за каналами, не взаимодействуя с ними. Наблюдатели обычно используются для журналирования чатов и текущей деятельности (например, входящих и исходящих звонков VoIP).
- *Согласователи (Approvers)*: Ответственны за предоставление пользователям возможности принимать или отклонять решение об использовании входящих каналов.
- *Обработчики (Handlers)*: Выполняют действительное взаимодействие с каналом. Это может быть подтверждение о получении текстового сообщения или отправка сообщений, отправка или получение файлов, и т.д. Обработчик, как правило, связан с пользовательским интерфейсом.

Клиентским программам предоставляются сервисы D-Bus с одним, двумя или тремя следующими интерфейсами: `Client.Observer`, `Client.Approver` и `Client.Handler`. В каждом интерфейсе есть метод, который может вызываться диспетчером каналов для того, чтобы проинформировать клиентскую программу о канале, за которым осуществляется наблюдение (т. е. будут выполнены операции, реализуемые с помощью клиентской программы типа `Observer`), относительно которого принимается решение о подтверждении или отклонения с ним работы(т. е. будут выполнены операции, реализуемые с помощью клиентской программы типа `Approver`) и который будет обрабатываться (т. е. будут выполнены операции, реализуемые с помощью клиентской программы типа `Handler`).

Диспетчер каналов, в свою очередь, осуществляет распределение канала по всем группам клиентских приложений. Сначала канал распределяется по всем соответствующим наблюдателям (клиентским программам типа `Observer`). После того, как поступят все ответы, канал будет распределен по всем соответствующим согласователям (клиентским программам типа `Approver`). После того, как первый согласователь одобрит или отклонит использование канала, об этом будут проинформированы все другие согласователи и канал, наконец, будет отправлен обработчику (клиентской программе типа `Handler`). Диспетчеризация канала происходит поэтапно, поскольку прежде, обработчик начнет изменять состояние канала, программам-наблюдателям может потребоваться время с тем, чтобы выполнить некоторые настройки.

У клиентских программ есть свойство — свойство фильтра канала, в котором перечисляются фильтры, считываемые диспетчером каналов с тем, чтобы знать, какие каналы интересуют данное клиентское приложение. В фильтре должны быть указаны, как минимум, тип канала и тип целевого хэндла (например, контакт или чат-комната), которые интересуют клиентскую программу, но в нем также можно указывать и другие свойства. Сравнение с неизменяемыми свойствами канала выполняется с помощью простой операции сравнения. В таблице 20.5 показан фильтр, который без всяких изменений должен присутствовать во всех каналах текстовых сообщений.

Рис.20.5: Пример фильтра канала

Свойство	Значение
<code>ofdT.Channel.ChannelType</code>	<code>Channel.Type.Text</code>
<code>ofdT.Channel.TargetHandleType</code>	<code>Handle_Type_Contact (1)</code>

Поиск клиентских приложений осуществляется через шину D-Bus, поскольку они публикуют сервисы, имя которых начинается с хорошо известного префикса `ofdT.Client` (например, `ofdT.Client.Empathy.Chat`). Они также могут установить файл, из которого диспетчер каналов прочитает фильтры. Это позволит диспетчеру каналов запустить клиентское приложение в случае,

если оно еще не запущено. То, что клиентские приложения можно искать таким образом, позволяет делать пользовательский интерфейс настраиваемым, который можно изменять в любой момент, не заменяя при этом каких-либо других частей фреймворка Telepathy.

Всё или ничего

Есть возможность задать фильтр, указывающий, что вам интересны все каналы, но на практике это используется только в качестве примера наблюдения за каналами. В реально существующих клиентских приложениях есть участки кода, которые специально предназначены для определенных типов каналов.

Пустой фильтр означает, что обработчика не интересуют никакие каналы. Не смотря на это, есть возможность передать канал такому обработчику, если вы сделаете это, указав имя обработчика. Такие фильтры используются во временно создаваемых обработчиках, которые создаются по запросу.

20.4. Роль привязок к языкам программирования

Поскольку фреймворк Telepathy является интерфейсом API для D-Bus, им можно управлять с помощью любого языка программирования, в котором поддерживается работа с шиной D-Bus. Для Telepathy не требуются привязки к языкам программирования, но привязками можно пользоваться с целью удобства в работе.

Языковые привязки можно разделить на две группы: низкоуровневые привязки, в которых есть код, сгенерированный с использованием спецификаций, констант, названий методов, и т.д., и высокоуровневые привязки, написанные программистами для того, чтобы облегчить другим программистам возможность что-либо делать с использованием фреймворка Telepathy. Примерами высокоуровневых привязок являются привязки к GLib и к Qt4. Примерами низкоуровневых привязок являются привязки к языку Python и привязки к языку C для библиотеки libtelepathy, входящей в состав фреймворка; впрочем, привязки к GLib и к Qt4 также относятся к низкоуровневым привязкам.

20.4.1. Асинхронное программирование

В языковых привязках все обращения к методам, с помощью которых осуществляются запросы кшине D-Bus, являются асинхронными: выполняется запрос, а ответ осуществляется через функцию обратного вызова (через callback). Это необходимо, поскольку сама шина D-Bus является асинхронной.

Как и в большинстве случаев программирования сетевых и пользовательских интерфейсов, программирование вшине D-Bus требует использовать цикл обработки событий с тем, чтобы выполнять диспетчеризацию обратных вызовов для входящих сигналов и результатов вызовов методов. Шина D-Bus хорошо интегрируется с главным циклом GLib, используемых в инструментальных наборах GTK+ и Qt.

В языковых привязках шины D-Bus (например, для dbus-glib) предоставляет псевдо-синхронный интерфейс API, в котором главный цикл блокируется до тех пор, пока метод не вернет результат. Когда-то это можно было делать с помощью привязок к API telepathy-glib. К сожалению, использование псевдо-синхронного интерфейса API привело к возникновению проблем и, в конце концов, эта возможность была убрана из telepathy-glib.

Почему не работают псевдо-синхронные обращения к шине D-Bus

Псевдо-синхронный интерфейс, предоставляемый `dbus-glib` и другими привязками D-Bus, реализован с использованием техники вида «запрос и блокирование». Когда устанавливается блокировка, только для сокета D-Bus выполняется опрос новых событий ввода-вывода, а все сообщения шины D-Bus, которые не являются ответом на запрос, помещаются в очередь для последующей обработки.

Это ведет к нескольким серьезным и неизбежным проблемам:

- Клиентское приложение, которое делает вызов, блокируется в ожидании ответа на запрос. Оно (а также его пользовательский интерфейс, если он имеется), полностью перестанет реагировать. Если запрос требует обращения к сети, то на это уйдет определенное время. Если вызываемый метод будет заблокирован, то тот, кто его вызвал, перестанет реагировать до тех пор, пока не наступит таймаут.

Использование потоков в данном случае также не будет решением, поскольку это всего лишь еще один способ сделать ваши вызовы асинхронным. Вместо этого вы можете с тем же самым успехом делать асинхронные вызовы, а ответы получать с помощью уже существующего цикла обработки событий.

- Порядок сообщений может быть изменен. Те сообщения, которые были получены до ожидаемого ответа, будут добавлены в очередь и доставлены в клиентское приложение только после получения ответа.

Это приводит к проблемам, когда сигнал, оповещающий об изменении состояния (например, что объекта, который должен быть уничтожен), будет получен уже после того, как вызов метода для этого объекта завершится ошибкой (он завершится исключением вида \code{UnknownMethod}). В данной ситуации сложно решить, какое сообщение об ошибке показывать пользователю. В случае же, когда мы сначала получаем сигнал, мы можем отменить приостановленные вызовы методов D-Bus, или проигнорировать их ответы.

- Два процесса, делающие псевдо-блокирующие вызовы друг-друга, могут попасть в состояние взаимной блокировки (deadlock), причем каждый в своей очереди будет ожидать ответа от другого процесса. Такой сценарий может произойти с процессами, которые оба являются сервисами D-Bus (например, клиентскими программами Telepathy). Диспетчер каналов вызывает методы в этих клиентских приложениях с тем, чтобы назначить им каналы, но клиентские приложения вызывают методы в диспетчере каналов с тем, чтобы запросить открытие новых каналов (или, аналогично, вызывают методы менеджера аккаунтов, который является частью того же самого процесса).

Вызовы методов в первых привязках Telepathy, сгенерированных для языка C, просто пользовались функцией обратного вызова для `typedef`. Ваша функция обратного вызова просто должна реализовывать сигнатуру того же самого типа.

```
typedef void (*tp_conn_get_self_handle_reply) (
    DBusGProxy *proxy,
    guint handle,
    GError *error,
    gpointer userdata
);
```

Эта идея проста, и работает для языка C, поэтому она продолжала использоваться в следующем поколении привязок.

В последние годы был разработан способ, позволяющий использовать такие скриптовые языки, как, например, Javascript и Python, и язык, похожий на язык C# и называющийся Vala, так, чтобы интерфейсами API, базирующимися на Glib/GObject, стало можно пользоваться через инструмент-

тальное средство, называемое GObject-Introspection. К сожалению, очень сложно сделать так, чтобы привязать использование обратных вызовов этих типов к другим языкам программирования. Поэтому новые привязки создавались такимим, чтобы они могли использовать возможности асинхронных обратных вызовов, предоставляемыми этими языками и библиотекой GLib.

20.4.2. Готовность объектов

В простом интерфейсе API для шины D-Bus, таком как низкоуровневые привязки фреймворка Telepathy, вы можете начать создавать вызовы методов или принимать сигналы для объекта просто с помощью создания для него прокси-объекта. Это столь же просто, как передать путь к объекту и имя интерфейса и произвести запуск.

Однако в высокоуровневом API фреймворка Telepathy нам нужно, чтобы в прокси-объектах наших объектов было известно, какие есть интерфейсы, нам нужно, чтобы можно было получать описание базовых свойств для данного типа объектов (например, тип канала, цель использования канала и кто был инициатором его открытия), а также нам необходимо определять и отслеживать состояние или статус объекта (например, статус соединения).

Таким образом, для всех прокси-объектов существует понятие готовности. С помощью вызова метода в прокси-объекте, вы можете асинхронно подготовить информацию о состоянии этого объекта и, при этом вы будете уведомлены, когда информация о состоянии объекта будет подготовлена, а это будет означать, что объект готов к использованию.

Поскольку не все клиентские программы реализованы и не все из них могут нас интересовать, все возможности данного объекта, подготавливаемые для объекта определенного типа, выделяются в ряд допустимых возможностей. В каждом объекте реализуется базовая возможность, в которой будет подготовлена наиболее важная информация об объекте (например, его свойство `Interfaces` и его начальное состояние), плюс ряд необязательных возможностей для дополнительного состояния, которое может включать в себя дополнительные свойства или возможность отслеживания состояния объекта. Конкретными примерами дополнительных возможностей, которые вы можете подготавливать на различных прокси, являются информация о контакте, свойства объекта, геолокационная информация, состояния чата (например, «Пользователь печатает сообщение...») и аватары пользователей.

Например, в прокси объектах соединений обычно имеется:

- базовая возможность (core feature), которая позволяют получать статус интерфейса и соединения,
- возможность получения информации о классах запрашиваемых каналов и поддерживаемых контактах и
- возможность установки соединение и получения сигнала о его готовности, когда оно будет установлено.

Программист запрашивает объект, который нужно будет подготовить, предоставляя список возможностей, которые ему интересны, и функцию обратного вызова, которая будет вызвана, когда эти возможности станут готовыми. Если все возможности уже подготовлены, то функция обратного вызова может вызываться сразу, в ином случае функция обратного вызова будет вызвана тогда, когда все указанные возможности будут найдены.

20.5. Устойчивость к ошибкам

Одним из ключевых преимуществ фреймворка Telepathy является его устойчивость к ошибкам (robustness). Компоненты являются модулями, поэтому проблемы в одном компоненте не должны

выводить из строя всю систему. Ниже перечисляются некоторые из особенностей фреймворка Telepathy, которые делают его устойчивым к ошибкам:

- Менеджер аккаунтов и диспетчер каналов могут восстанавливать свое состояние. Когда запускается центр управления (единственный процесс, в состав которого входит менеджер аккаунтов и диспетчер каналов), он просматривает имена сервисов, зарегистрированных в текущий момент на сессионнойшине пользователя. Все соединения, которые он найдет и сможет проассоциировать с известным аккаунтом, будут привязаны к этому аккаунту (вместо того, чтобы устанавливать новое соединение), а из работающих клиентских программ будет получен список каналов, которые они обрабатывают.
- Если клиентская программа пропадает, а канал, который она обрабатывает, остается открытый, то диспетчер каналовerezапустит ее клиентскую программу и передаст ей канал.
- Если клиентская программа повторно выходит из строя, диспетчер каналов может попытаться запустить другую клиентскую программу, если таковая имеется, или закрыть канал (чтобы предотвратить повторный выход клиентской программы из строя на тех данных, которые она не способна обработать).

Текстовые сообщения, прежде чем они исчезнут из списка ожидающих сообщений, требуют подтверждения. Подразумевается, что клиентская программа подтвердит получение сообщения как только она будет уверена, что пользователь увидел это сообщение (то есть оно было показано в активном окне). Таким образом, если клиентская программа выйдет из строя, пытаясь выдать сообщение, сообщение останется в канале в очереди ожидающих сообщений как ранее не показанное.

- Если из строя выйдет соединение, менеджер аккаунтовerezапустит его. Очевидно, что содержимое любых каналов, у которых есть состояние (*stateful*), будет потеряно, но это повлияет только на соединения, запущенные в данном процессе, и не повлияет на другие. Клиентские программы могут следить за состоянием соединений и просто повторноerezапрашивать информацию, например, реестр контактов, а также не повлияет на те каналы, у которых нет состояния (*stateless*).

20.6. Расширение Telepathy: механизм *sidecars*

Не смотря на то, что спецификация Telepathy старается охватить широкий спектр возможностей, предоставляемых коммуникационными протоколами, некоторые протоколы сами являются расширяемыми [4]. Разработчикам фреймворка Telepathy хотели сделать так, чтобы можно было использовать такие расширения без необходимости расширения самой спецификации Telepathy. Это было сделано с помощью механизма *sidecars*.

Механизм *sidecars* обычно реализуется с помощью плагинов менеджера соединений. Клиентская программа вызывает метод, запрашивающий *sidecar*, в котором реализован данный интерфейс шины D-Bus. Например, в чьей-то реализации списков приватности XEP-0016 может быть реализован интерфейс с названием `com.example.PrivacyLists`. Затем метод вернет объект D-Bus, предоставленный плагином, который должен реализовывать этот интерфейс (и, возможно, другие интерфейсы). Объект существует вместе с основным объектом соединения (наподобие мотоциклетной коляски - «*sidecar*», которая крепится сбоку мотоцикла).

История механизма sidecars

В самом начале разработки фреймворка Telepathy, в проекте One Laptop Per Child («Каждому ребенку свой ноутбук») для совместного обмена информацией между устройствами потребовалась поддержка своих собственных расширений протокола XMPP (XEPs). Они были добавлены прямо в Telepathy-Gabble (менеджер соединений для XMPP) и предоставлялись через недокументированные интерфейсы объекта соединения. В конце концов, всё больше и больше разработчиков хо-

тело поддерживать конкретные расширения XEP, для которых не было аналогов в других коммуникационных протоколах. Было решено, что для плагинов необходим новый, более общий интерфейс.

20.7. Беглый взгляд внутрь менеджера соединений

Большинство менеджеров соединений написаны с использованием языковых привязок к C/GLib, причем был разработан ряд высокогенеральных базовых классов, которые делают более простым написание менеджеров соединений. Как упоминалось ранее, объекты D-Bus публикуются из объектов программы, в которых реализован ряд интерфейсов, отображаемых в интерфейсы D-Bus. В библиотеке Telepathy-GLib представлены базовые объекты для реализации объектов менеджеров соединений, соединений и каналов. В этой библиотеке предлагается интерфейс реализации менеджера каналов. Менеджеры каналов являются фабриками, которые можно использовать для того, чтобы с помощью `BaseConnection` получать экземпляры объектов каналов и управлять ими при публикации их на шине.

В привязках также предоставляется то, что называется *миксинами* (mixins). Их можно добавлять к классу с тем, чтобы с помощью одного и того же механизма предоставлять дополнительные функциональные возможности, абстрагировать спецификации API, а также обеспечивать обратную совместимость между новыми и устаревшими версиями API. Самым распространенным примером использования является mixin, который добавляет к объекту интерфейс свойств D-Bus. Также есть mixins для реализации интерфейсов `ofdT.Connection.Interface.Contacts` и `ofdT.Channel.Interface.Group` и mixins, позволяющие с помощью одного и того же набора методов реализовывать старые и новые интерфейсы присутствия и старые и новые интерфейсы поддержки текстовых сообщений.

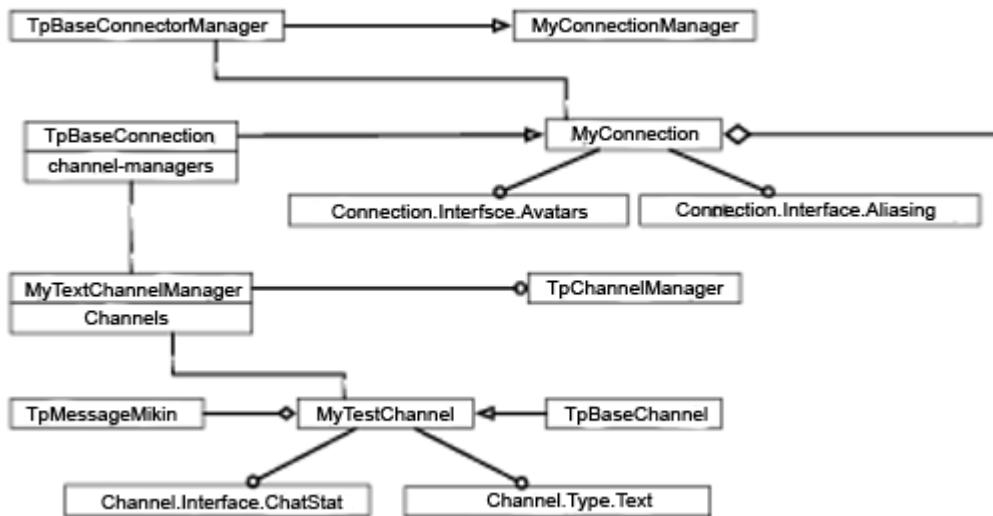


Рис.20.5: Пример архитектуры менеджера соединений

Использование миксинов для решения проблем с ошибкой в API

Одним из мест, где mixins были использованы для исправления ошибки в спецификации Telepathy, является `TpPresenceMixin`. Исходный интерфейс, предложенный в Telepathy (`ofdT.Connection.Interface.Presence`), был сильно пере усложнен, его было сложно использовать при реализации соединений и клиентских программ, и в нем была представлена та функциональность, которая отсутствовала в большинстве коммуникационных протоколах, а в остальных - использовалась редко. Этот интерфейс был заменен более простым интерфейсом (`ofdT.Connection.Interface.SimplePresence`), в котором была представлена функциональность, необходимая пользователям, и которая действительно была реализована в менеджерах соединений.

В миксине присутствия реализованы оба интерфейса соединения, позволяющие старым клиентским приложениям продолжать работать, но на функциональном уровне более простого интерфейса.

20.8. Усвоенные уроки

Фреймворк Telepathy является превосходным примером того, как можно поверх D-Bus создать модульный и гибкий интерфейс API. Он показывает, как нужно поверх D-Bus разрабатывать расширяемый фреймворк, состоящий из нескольких слабо связанных между собой частей. Причем такой, для которого не нужен центральный управляющий демон, и в котором отдельные компоненты можно запускать независимо друг от друга и без потери данных в других компонентах. Telepathy также показывает, как можно эффективно использовать шину D-Bus, минимизируя объем трафика, передаваемого через шину.

Разработка Telepathy была итеративной, постепенно улучшавшей использование шины D-Bus. Были ошибки и были усвоены уроки. Далее перечисляется наиболее важное из того, что мы усвоили, когда разрабатывали архитектуру Telepathy:

- *Использование свойств шины D-Bus; чтобы получить информацию, не нужно вызывать десятки методов D-Bus.* Для каждого вызова метода для передачи данных туда и обратно требуется некоторое время. Вместо того, чтобы делать массу отдельных вызовов (например, `GetHandle`, `GetChannelType`, `GetInterfaces`), используйте свойства шины D-Bus и получайте всю информацию с помощью одного вызова `GetAll`.
- *Когда объявляете о создании нового объекта, то предоставляйте столько информации, сколько вы можете предоставить.* Первое, что обычно делают клиенты, когда узнают о новом объекте, это делают запрос всех свойств объекта, чтобы решить, нужен ли им вообще данный объект. Если включить все неизменяемые свойства объекта в сигнал, оповещающий о создании объекта, то большинство клиентов сможет определить, насколько им интересен данный объект, не делая при этом никаких вызовов методов. Более того, если им интересен объект, им не нужно беспокоить его запросами о неизменяемых свойствах.
- *Интерфейс Contacts позволяет запрашивать информацию сразу из нескольких интерфейсов.* Вместо того, чтобы получать всю информацию о контакте с помощью большое количество вызовов `GetAll`, вы можете с помощью интерфейса `Contacts` запросить всю информацию за один раз и, при этом, сэкономить большой объем трафика, передаваемого через шину D-Bus.
- *Не используйте абстракции, которые не совсем подходят.* Предоставление реестра контактов и групп контактов в виде каналов, реализующих интерфейс `Group`,казалось неплохой идеей, т.к. для этого использовались существующие абстракции, а не добавлялись новые интерфейсы. Однако, это сделало реализацию клиентских приложений сложной, что, в конце концов, перестало нам подходит.
- *Убедитесь, что ваш интерфейс API будет в будущем соответствовать вашим потребностям.* Первоначальный интерфейс API запроса каналов оказался недостаточно гибким и позволял делать лишь весьма общие запросы. Это нам не подошло, когда нам понадобилось делать запросы в каналы для получения большего количества информации. Этот интерфейс API пришлось заменить другим, который был намного более гибким.

Примечания

1. <http://telepathy.freedesktop.org/> или смотрите руководство разработчика на <http://telepathy.freedesktop.org/doc/book/>
2. <http://telepathy.freedesktop.org/spec/>
3. Далее мы будем сокращать `/org/freedesktop/Telepathy/` и `org.freedesktop.Telepathy` как `\code{ofdT}` с целью экономии места.

4. Например, XMPP (Extensible Messaging and Presence Protocol, т.е. расширяемый протокол обмена сообщениями и информацией о присутствии.)

[На главную](#) -> [MyLDP](#) -> [Тематический каталог](#) ->

21. Фреймворк Thousand Parsec

Глава 21 из 1 тома книги ["Архитектура приложений с открытым исходным кодом"](#).

Огромная звездная империя включает в себя сотни миров, простираясь в космосе на тысячи парсеков. В отличие от некоторых других областей галактики, здесь живет несколько воинов; это люди с высоким интеллектом, с богатыми культурными и научными традициями. Их великолепные планеты, созданные на базе великих научных и технических достижений, являются лучом света для всех других миров этого этапа развития и процветания. Космические корабли прибывают из всего огромного квадранта, простирающегося во всех направлениях, и из-за его пределов, привозя отовсюду самых знаменитых исследователей. Они прибывают сюда с тем, чтобы внести свой вклад в самый амбициозный проект из тех, которые когда либо пытались создать разумные существа: создать децентрализованную компьютерную сеть для всей Галактики, со всеми ее различными языками, культурами и правовыми системами.

Thousand Parsec это больше, чем видеоигра: это фреймворк с полным набором инструментальных средств для создания пошаговых многопользовательских космических игр-стратегий. Его основной игровой протокол позволяет применять разнообразные клиентские программы, серверы и интеллектуальное программное обеспечение, а также использовать все это в широком спектре различных игр. Хотя размер фреймворка потребовал планирования и дополнительных исследований, заставивших участников разработки держаться тонкой грани между чрезмерным развитием системы в вертикальном и в горизонтальном направлениях, он также сделал фреймворк довольно интересным образцом для обсуждения архитектуры приложений с открытым исходным кодом.

Журналисты играм жанра Thousand Parsec в соответствие с тем, как должен действовать игрок, управляющий империей, присвоили термин «4X» - сокращение от «explore, expand, exploit, and exterminate» («исследовать, развивать, эксплуатировать и уничтожать»). Обычно в играх жанра 4X игроки должны изучить карту (исследовать), создать новые поселения или расширить влияние существующих (развивать), собирать и использовать ресурсы в районах, находящихся под их контролем (эксплуатировать), и атаковать и устранять игроков-конкурентов (уничтожать). Акцент на экономическом и технологическом развитии, локальном управлении и разнообразии путей к господству придают жанру игр больших стратегий беспрецедентную глубину и сложность.

С точки зрения игрока, в игре Thousand Parsec есть три основных компонента. Во-первых, клиентская программа: это приложение, через которое игрок взаимодействует со Вселенной. Она с помощью всех важных протоколов подключается через сеть к серверу, к которому также подключены другие клиентские программы (или, в некоторых случаях, искусственный интеллект). На сервере хранится состояние всей игры, обновляемое клиентскими программами в начале каждого хода. Игроки могут выполнять различные действия и передавать их обратно на сервер, который вычисляет результирующее состояние для следующего хода. Характер действий, которые игрок может выполнить, определяется набором правил: они, по сути, определяются игрой, в которую играют, которая актуализирована для игрока при помощи любой поддерживаемой клиентской программы.

Благодаря разнообразию возможных игр и сложности архитектуры, которая необходима для поддержки такого разнообразия, фреймворк Thousand Parsec стал интересным проектом как для геймеров, так и для разработчиков. Мы надеемся, что даже серьезный кодировщик, немного интересующийся анатомией игрового фреймворка, может найти для себя ценное в механике, лежащей в основе клиент-серверного взаимодействия, динамическом конфигурировании, обработке метадан-

ных и реализаций слоев, во всем, что с течением лет естественным образом переросло в хороший проект, являющийся квинтэссенцией стиля открытого исходного кода.

В своей основе фреймворк Thousand Parsec является, в первую очередь, набором стандартных спецификаций игрового протокола и других связанных с этим функций. В данной главе фреймворк рассматривается, в основном, с этой абстрактной точки зрения, но во многих случаях гораздо более поучительно обратиться к фактической реализации. С этой целью авторы для конкретного обсуждения выбрали «флагманские» реализации каждого из основных компонентов.

Образцом клиентской программы является `tpclient-pywx`, достаточно совершенная клиентская программа на базе wxPython, которая в настоящее время поддерживает самый большой набор функций и новейшую версию игрового протокола. Это поддерживается с помощью клиентской библиотеки языка Python `libtpclient-py`, предоставляющей средства кэширования и другие функции, а также с помощью библиотеки Python `libtppproto-py`, в которой реализована последняя версия протокола Thousand Parsec. Что касается сервера, то образцом является `tpserver-cpp` - совершенная реализация на языке C++, поддерживающая самые новые функции и последнюю версию протокола. Этот сервер отрабатывает многочисленные наборы правил, среди которых образом наиболее широкого использования возможностей «традиционной» космической игры жанра 4X и важным этапным событием является набор правил *Missile and Torpedo Wars* (Ракетные и торпедные войны).

21.1. Анатомия звездной империи Star Empire

Чтобы познакомить с тем, что составляет универсум Thousand Parsec, имеет смысл сначала кратко рассказать об игре. Для этого мы рассмотрим набор правил *Missile and Torpedo Wars* (Ракетные и торпедные войны), второй важный набор правил проекта, в котором используется большинства основных функций текущей версии основного протокола Thousand Parsec. Некоторые используемые здесь термины еще не знакомы; для того, чтобы все встало на свои места, в оставшейся части этого раздела будут приведены их объяснения.

Missile and Torpedo Wars (Ракетные и торпедные войны) является расширенным набором правил в том смысле, что в нем реализованы все методы, имеющиеся во фреймворке Thousand Parsec. На момент написания статьи, это единственный набор правил, в котором это сделано, и этот набор правил быстро расширяется с тем, чтобы стать более полноценной и развлекательной игрой.

Когда устанавливается соединение с сервером Thousand Parsec, клиентская программа запрашивает с сервера список игровых объектов и переходит к загрузке всего каталога. В этом каталоге находятся все объекты, форумы, сообщения, категории, модели, компоненты, свойства, игроки и ресурсы, определяющие состояние игры и которые подробно описываются в данном разделе. Хотя может показаться, что этой информации слишком много с тем, чтобы клиентская программа могла справиться с ней в начале игры, а также в конце каждого хода, эта информация является для игры абсолютно необходимой. Как только эта информация будет загружена, что обычно занимает порядка нескольких секунд, в клиентской программе теперь будет все, что нужно, чтобы преобразовать эту информацию в свое собственное представление универсума игры.

При первом подключении к серверу случайным образом генерируется Планета, которая определяется как «домашняя Планета» нового игрока, и на ней автоматически создаются две Флотилии. Каждая Флотилия по умолчанию состоит из двух Кораблей-разведчиков, состоящих из разведывательной капсулы с ракетной пусковой установкой Alpha. Поскольку боеприпасов еще нет, то такая Флотилия, созданная по умолчанию, еще не может вести бой с другой Флотилией или с Планетой; это по сути то, где зарыта собака.

В этот момент для игрока очень важно начать оснащение Флотилии оружием. Это достигается при помощи создания распоряжения на сборку оружия *Build Weapon order*, а затем с помощью распо-

ряжения на загрузку вооружения Load Armament order отгрузки готовой продукции для целевой Флотилии. Распоряжение на сборку оружия преобразует ресурсы Планеты, количество которых и их пропорциональное соотношение установлено для каждой Планеты случайным образом, в готовый продукт: в боеприпасы, которые создаются из ресурсов на поверхности Планеты. Затем распоряжение на загрузку вооружения перенаправляет это созданное вооружение для ожидающей Флотилии.

После того, как будут использованы легкодоступные ресурсы, находящиеся на поверхности Планеты, для того, чтобы получить их, еще потребуется их добыча в шахтах. Ресурсы бывают в двух различных состояниях: извлекаемые и неизвлекаемые. При помощи передачи распоряжения на Планету на добычу ресурсов в шахтах (Mine order), извлекаемые ресурсы могут в течение достаточно длительного времени преобразовываться в поверхностные ресурсы, которые затем могут быть использованы для производства.

21.1.1. Объекты

В универсуме Thousand Parsec каждая физическая вещь является объектом. На самом деле, сам универсум также является объектом. Такая конструкция позволяет иметь практически неограниченный набор элементов в игре, оставаясь при этом простой для набора правил, для которого нужны только несколько типов объектов. На верхнем уровне, где добавляются новые типы объектов, каждый объект может хранить некоторую свою собственную информацию, которую можно пересыпать и использовать через протокол Thousand Parsec. В настоящее время по умолчанию предоставлены пять основных встроенных типов объектов: Вселенная (Universe), Галактика (Galaxy), Звездная система (Star System), Планета (Planet) и Флотилия (Fleet).

Вселенная является в игре Thousand Parsec объектом верхнего уровня и она всегда доступна для всех игроков. Хотя объект Вселенная, на самом деле, сильно не влияет на управление игрой, в нем хранится очень важная информация: номер текущего шага. Номер шага, также известный на жаргоне Thousand Parsec как "год", после завершения каждого хода увеличивается, естественно, на единицу. Он хранится в виде беззнакового 32-разрядного целого числа, что позволяет запускать игры вплоть до 4294967295 года. Авторы, на сегодняшний день, не видели, чтобы процесс игры заходил так далеко, хотя, в теории, это не исключено.

Галактика является контейнером для ряда более уточненных объектов - Звездных систем, Планет и Флотилий и не предоставляет никакой дополнительной информации. В игре может существовать большое количество Галактик, причем каждая из них является подчастью Вселенной.

Как и два предыдущих объекта, Звездная система является, прежде всего, контейнером для объектов следующего более низкого уровня. Но объект Звездная система является первым уровнем объектов, который в клиенте представлен графически. Эти объекты могут содержать в себе Планеты и Флотилии (по крайней мере, временно).

Планета является большим небесным телом, которое может быть заселено и может иметь шахты для добычи ресурсов, производственные мощности, наземное вооружение и многое другое. Планеты являются первым уровнем объектов, которые могут принадлежать игроку; владение Планетой является достижением и этого добиться нелегко, а отсутствие во владении каких-либо Планет является в наборе правил типичным условием провозгласить поражение игрока. Объект Планета имеет относительно большой объем сохраняемых данных, среди которых есть следующее:

- Идентификатор игрока владельца Планеты (или -1, если Планета никому не принадлежит).
- Список ресурсов Планеты, содержащий идентификатор ресурса (тип) и количество этого ресурса на Планете на ее поверхности, количество, которое можно добыть в шахтах, и количество недоступных ресурсов этого типа.

Встроенные объекты, перечисленные выше, представляют собой хорошую базу для многих наборов правил, соответствующих традиционной формуле игр космических стратегий жанра 4Х. Естественно, что в соответствие с принципами инженерии хорошего программного обеспечения, внутри набора правил классы объектов можно расширять. Таким образом, разработчик набора правил имеет возможность создавать новые типы объектов или хранить в существующих типах объектов дополнительную информацию, соответствующую тому, что требуется в наборах правил, т.е. предоставляется возможность фактически неограниченного расширения физических объектов, имеющихся в игре.

21.1.2. Распоряжения

В соответствии с каждым набором правил, распоряжения могут направляться как объектам Флотилия, так и объектам Планета. Хотя главный сервер не поставляется с какими-либо типами распоряжений, определяемыми по умолчанию, они как раз являются существенной частью основы игры. В зависимости от характера набора правил, распоряжения могут быть использованы для выполнения практически любой задачи. В духе жанра 4Х есть несколько стандартных распоряжений, которые реализованы в большинстве наборов правил: это следующие распоряжения — Переместиться (Move), Перехватить (Intercept), Построить (Build), Колонизировать (Colonize), Добывать (Mine) и Атаковать (Attack).

Чтобы выполнить первый императив (исследовать) жанра 4Х, нужно иметь возможность перемещаться по карте Вселенной. Это обычно достигается с помощью распоряжения Переместиться, которое передается объекту Флотилия. В гибком и расширяемом по духу фреймворке Thousand Parsec распоряжение Переместиться может быть реализовано по-разному в зависимости от природы набора правил. В наборах правил *Minisec* и *Missile and Torpedo Wars*, распоряжение Переместиться обычно в качестве параметра использует точку в трехмерном пространстве. На стороне сервера вычисляется расчетное время прибытия и клиентской программе отсылается количество требуемых шагов. Распоряжение Переместиться также действует как псевдо-распоряжение Атаковать в тех наборах правил, где не реализована командная работа. Например, перемещение к точке, которую занимает вражеская Флотилия, в обоих наборах правил *Minisec* и *Missile and Torpedo Wars* почти наверняка означает, что затем последует период интенсивных боевых действий. В некоторых наборах правил, в которых поддерживается распоряжение Переместиться, применяется другая параметризация (т.е. не используются точки трехмерного пространства). Например, в наборе правил *Risk* допускается только одношаговый переход к Планетам, соединенных между собой "туннелем".

Распоряжение Перехватить обычно отправляется объекту Флотилия, что позволяет встретиться в космосе с другой Флотилией (обычно вражеской). Это распоряжение похоже на распоряжение Переместиться, но поскольку во время, пока выполняется шаг игры, два объекта могут двигаться в разных направлениях, невозможно, указав только координаты в пространстве, встретиться непосредственно с другой Флотилией, поэтому необходим другой тип распоряжения. Эту проблему решает распоряжение Перехватить и его можно использовать для уничтожения неприятельской Флотилии в глубоком космосе или для того, чтобы задержать предстоящую атаку в момент кризиса.

Распоряжение Построить помогает выполнить два императива жанра 4Х — развивать и эксплуатировать. Очевидным средством экспансии во Вселенной является создание большого количества Флотилий и затем Перемещения их вдаль и вширь. Распоряжение на Постройку, как правило, направляется объектам Планета и часто ограничивается количеством ресурсов, которые есть на Планете, и тем, как они используются. Если игроку повезло иметь домашнюю Планету, достаточно богатую ресурсами, то этот игрок может при Постройке кораблей достаточно рано в игре получить преимущество.

Точно также, как распоряжение Построить, распоряжение Колонизировать помогает выполнять императивы развивать и эксплуатировать. Распоряжение Колонизировать, почти всегда добавляе-

мое к объектам Флотилия, позволяет игроку присвоить себе ничейные Планеты. Это помогает расширять контроль над Планетами по всей Вселенной.

Распоряжение Добывать ресурсы в шахтах воплощает в себе императив эксплуатировать. Это распоряжение, как правило, направляемое объектам Планета и другим небесным телам, позволяет игроку добывать неиспользованные ресурсы, которые не сразу доступны на поверхности. В результате эти ресурсы переносятся на поверхность, что позволяет в дальнейшем использовать их для построения Флотилий и, в конечном, итоге расширять экспансию игрока во Вселенной.

Распоряжение Атаковать, реализованное в некоторых наборах правил, позволяет игроку явно инициировать бой с Флотилией или Планетой противника, выполняя тем самым последний императив жанра 4X (уничтожить). В наборах правил, допускающих командную игру, команде важно иметь различные варианты распоряжений Атаковать (в отличие от простого использования распоряжений Переместиться и Перехватить при неявной атаки целей) с тем, чтобы избежать огня по дружественным игрокам или иметь возможность координировать совместную атаку.

Поскольку для фреймворка Thousand Parsec необходимы разработчики наборов правил, которые определяют свои собственные типы распоряжений, можно, и даже желательно, мыслить нестандартно и создавать свои собственные распоряжения, которых еще нигде нет. Возможность добавлять дополнительные данные к любому объекту позволяет разработчикам делать со своими типами распоряжений очень интересные вещи.

21.1.3. Ресурсы

Ресурсы являются дополнительными элементами данных, которые упакованы в объекты игры. Широко используемые, особенно в объектах Планета, ресурсы позволяют легко расширять наборы правил. Как и многими другими проектными решениями, применяемыми в Thousand Parsec, расширяемость была движущим фактором при добавлении понятия ресурсы.

Хотя ресурсы, как правило, добавляются разработчиками набора правил, есть один ресурс, который можно использовать во фреймворке везде: ресурс Домашняя Планета (Home Planet), который используется для идентификации родной Планеты игрока.

Согласно практике использования фреймворка Thousand Parsec, ресурсы, как правило, используются для представления того, что может быть преобразовано в некоторый тип объекта. Например, в игре *Minisec* реализован ресурс Части корабля (Ship Parts), количественное значение которого случайным образом присваивается каждому объекту Планета во Вселенной. Когда одна из таких Планет колонизирована, вы можете с помощью распоряжения Построить преобразовать такой ресурс в реально действующие Флотилии.

В игре *Missile and Torpedo Wars*, пожалуй, самое широкое использование ресурсов среди всех наборов правил, имеющихся на сегодняшний день. Это первый набор правил, где вооружение разрешено перемещать, что означает, что его можно добавлять на корабль с Планеты, а также удалить из корабля и добавлять обратно на Планету. Чтобы это можно было использовать, в игре для каждого вида оружия, созданного в игре, создается тип ресурсов. В результате на кораблях можно идентифицировать тип вооружения по его ресурсу, и перемешать вооружение по всей Вселенной. В *Missile and Torpedo Wars* с помощью ресурса Завод (Factory), назначенного каждой Планете, также отслеживается размещение заводов (производственные мощности Планет).

21.1.4. Модели

В фреймворке Thousand Parsec, как вооружение, так и корабли могут состоять из различных компонентов. Эти компоненты объединяются с тем, чтобы сформировать основу Модели (Design) - прототипа чего-нибудь такого, что можно построить и использовать в игре. Когда создается набор правил, разработчик практически сразу должен принять немедленное решение: должен ли набор

правил позволять динамически создавать модели вооружения и кораблей, или просто использовать список предопределенных моделей. С одной стороны, игры с использованием готовых моделей будет легче развивать и в них легче оценивать баланс ресурсов, но, с другой стороны, динамическое создание моделей добавляет в игру новый уровень сложности, новые проблемы и больше удовольствия.

Создаваемые пользователями модели позволяют сделать игру более усовершенствованной. Поскольку пользователи должны опираться на стратегии при разработке своих собственных кораблей и их вооружения, в игру добавляется пласт вариабельности, который может помочь еще одним способом смягчить большие преимущества, получаемые игроком благодаря удачным стечениям обстоятельств (например, при распределении Планет) или в результате других аспектов игровой стратегии. Управление этими моделями осуществляется в соответствии с вполне конкретными для каждого набора правил правилами, определяемыми для каждого компонента и записываемыми на языке компонентов Thousand Parsec Component Language (языке TPCL, который описан далее в этой главе). Получается так, что для реализации моделей вооружения и кораблей со стороны разработчиков не требуется дополнительного программирования каких-либо функций; достаточно настройки некоторых простых правил, которые для каждого компонента имеются в наборе правил.

Без тщательного планирования и надлежащего баланса возможностей, большое преимущество от использования своих собственных моделей может оказаться гибельным. На более поздних стадиях игры, на разработку новых типов оружия и кораблей при их создании может затрачиваться огромное количество времени. Также проблемы могут быть связаны с отсутствием достаточного опыта, необходимого на стороне клиентской программы для манипуляциями с моделями. Несмотря на то, что использование моделей может быть неотъемлемой частью одной игры, и, одновременно, не иметь никакого отношения к другой игре, значительным препятствием может оказаться добавление в клиентские программы окон, позволяющих обращаться к моделям. В клиенте `tpclient-pwux`, который является наиболее полным клиентом фреймворка Thousand Parsec, в настоящее время есть лаунчер этого окна, расположенный в сравнительно отдаленном месте - в подменю панели меню (который, в остальных случаях, в игре используется редко).

Функциональность Моделей реализована так, что они легко доступны для разработчиков наборов правил, что позволяет расширять игры практически до неограниченного уровня сложности. Во многих из существующих наборов правил разрешается использовать только предопределенные Модели. Однако, в *Missile and Torpedo Wars* можно из различных компонентов полностью создавать Модели вооружения и кораблей.

21.2. Протокол Thousand Parsec

Можно сказать, что протокол Thousand Parsec является той основой, на которой в проекте построено все остальное. В нем определены функции, доступные для тех, кто пишет наборы правил, серверы, которые должны работать, и то, с чем будут в состоянии справиться клиентские программы. Самое главное, что он, точно также как стандарт межзвездных сообщений, позволяет различным компонентам программного обеспечения понимать друг друга.

Сервер управляет фактическим состоянием и динамикой игры в соответствии с инструкциями, указываемыми в наборе правил. На каждом шаге клиентская программа игрока получает некоторую информацию о состоянии игры: объекты, кто ими владеет, распоряжения, которые выполняются, запасы ресурсов, состояние разработок, сообщения и все остальное, что может просматривать конкретный игрок. Игрок может выполнять определенные действия с учетом текущего состояния, например, отсылать распоряжения или создавать модели, а также отправлять их обратно на сервер для обработки при вычислении следующего шага. Все это взаимодействие оформлено в виде протокола Thousand Parsec. Интересным и вполне сознательным эффектом использования подобной архитектуры является то, что интеллектуальные клиентские программы, которые внешние по отношению к серверу / набору правил и являются лишь средством имитации в игре искус-

ственных компьютерных игроков, функционируют по тем же самым правилам, что и клиентские программы, управляемые игроками-людьми. Т.е. они не смогут "обманывать" за счет получения недопустимого доступа к информации или за счет возможности нарушать правила.

В спецификации протокола описывается последовательность фреймов, которые иерархические в том смысле, что каждый фрейм (за исключением фрейма заголовка Header) имеет базовый тип фрейма, к которому добавляются собственные данные конкретного фрейма. Имеется много абстрактных типов фреймов, которые никогда явно не использовались, а просто присутствуют для описания базы конкретных фреймов. Фреймы также могут быть односторонними, т. е. такие фреймы нужны для поддержки отсылки данных одной стороной (сервером или клиентской программой) и их получения другой.

Протокол Thousand Parsec предназначен для автономного использования поверх TCP/IP или туннелирования через другой протокол, например, HTTP. В нем также поддерживается шифрование SSL.

21.2.1. Основные фреймы

В протоколе предлагается несколько базовых фреймов, которые при взаимодействии между клиентской программой и сервером используются повсеместно. Ранее упомянутый фрейм Header (Заголовок) просто с помощью своих собственных двух прямых потомков — Request (Запрос) и Response (Ответ) представляет собой базу для создания всех остальных фреймов. Первый является основой для фреймов, с помощью которых соединение инициируется (в любом направлении), а второй - для фреймов, которые завершают соединение. Фреймы `ok` (Выполнено) и `fail` (Отказано), оба являются фреймами типа Response (Ответ), возвращающими при обмене данными два значения булевой логики. Фрейм Sequence (Последовательность), который также относится к типу Response (Ответ), указывает получателю, что в ответ на его запрос должны последовать несколько фреймов.

В протоколе Thousand Parsec для адресации объектов используются числовые идентификаторы. Соответственно, есть словарь фреймов, используемый при работе с данными через эти идентификаторы. Фрейм `Get With ID` (Получить с идентификатором) является базовым запросом данных по такому идентификатору; также есть фрейм `Get With ID and Slot` (Получить с идентификатором из слота) для данных, которые находятся в "слоте" родительских данных (объект — предок), имеющих идентификатор ID (например, распоряжение по объекту). Конечно, часто бывает нужно получать последовательности идентификаторов, такие как при первоначальном заполнении состояния клиента; это делается с помощью запроса типа `Get ID Sequence` (Получить последовательность идентификаторов) и ответов типа `ID Sequence` (Последовательность идентификаторов). Общая структура запроса нескольких элементов представляет собой запрос `Get ID Sequence` (Получить последовательность идентификаторов) и ответ `ID Sequence` (Последовательность идентификаторов), за которыми следует серия запросов `Get With ID` (Получить с идентификатором) и соответствующих ответов, описывающих требуемые данные.

21.2.2. Игроки и игры

Прежде, чем клиентская программа сможет начать взаимодействие с игрой, нужно соблюсти некоторые формальности. Клиентская программа должна сначала передать на сервер фрейм `Connect` (Подключение), на который сервер может ответить `ok` (Выполнено) или `fail` (Отказано), т.к. во фрейме `Connect` (Подключение) указывается версия протокола, используемого клиентской программой, и одной из причин отказа может быть несоответствие версий. Сервер может также ответить фреймом `Redirect` (Перенаправление) в случае перенаправления или использования серверных пулов. Далее, клиентская программа должна отправить фрейм `Login` (Регистрация), с помощью которого происходит идентификация и, возможно, аутентификация игрока; новые игроки на

сервере могут сначала использовать фрейм `Create Account` (Создать аккаунт) в случае, если сервер это позволяет.

Из-за огромной вариабельности протокола Thousand Parsec, клиентская программа должна каким-то образом выяснить, какие особенности протокола поддерживаются сервером; это осуществляется с помощью запроса `Get Features` (Получить описание возможностей) и ответа `Features` (Возможности). Среди некоторых особенностей, присутствующих в ответе, могут указываться следующие:

- Наличие туннелирования SSL и HTTP (для данного порта или другого порта).
- Поддержка расчета свойств компонента на серверной стороне.
- Упорядочивание последовательностей идентификаторов в ответах (по возрастанию или убыванию).

Аналогичным образом с помощью запроса `Get Games` (Получить игры) и последовательности ответов `Game` (Игра), клиентская программа будет проинформирована о том, какие игры активны на сервере. В отдельном фрейме `Game` (Игра) содержится следующая информация об игре:

- Длинное (описательное) название игры.
- Список поддерживаемых версий протокола.
- Тип и версия сервера.
- Название и версия набора правил.
- Список возможных конфигураций сетевых соединений.
- Несколько дополнительных элементов (количество игроков, количество объектов, особенности администрирования, комментарий, номер текущего шага и т.д.).
- Базовый URL для медиаресурсов, используемых в игре.

Для игрока, конечно, важно знать, против кого он играет (или, может быть, в зависимости от обстоятельств, вместе с кем он играет), и для этого имеется набор фреймов. Обмен осуществляется с использованием общепринятой схемы последовательности элементов с помощью запроса `Get Player IDs` (Получить идентификаторы игроков) и серии запросов `Get Player Data` (Получить данные об игроке) и ответов `Player Data` (Данные об игроке). Во фрейме `Player Data` (Данные об игроке) указывается имя и раса игрока.

Участие в игре также контролируется через протокол. Когда игрок закончит выполнение действий, он может проинформировать о готовности к следующему шагу с помощью запроса `Finished Turn` (Шаг закончен); переход на следующий шаг вычисляется после того, как это сделают все игроки. Переход на следующий шаг ограничен по времени, установленном на сервере, так что игроки, которые отвечают медленно или не отвечают, не могут участвовать в игре; клиентская программа обычно делает запрос `Get Time Remaining` (Получить оставшееся время) и по локальному таймеру отслеживает время, установленное в соответствие с ответом сервера `Time Remaining` (Оставшееся время).

Наконец, в протоколе Thousand Parsec поддерживается работа с сообщениями, предназначенными для различных целей: передаче сообщений для всех игроков, игровых уведомлений для отдельного игрока, связи между отдельными игроками. Есть варианты, которые организованы в виде контейнеров "форум", в которых можно указывать порядок и видимость сообщений; последовательность элементов обмена сообщений, которая следует далее, состоит из запроса `Get Board IDs` (Получить идентификаторы форумов), ответа `List of Board IDs` (Список идентификаторов форумов) и серии запросов `Get Board` (Получить форум) и ответов `Board` (Форум).

Как только клиентская программа получит информации, что для нее на форуме есть сообщение, она может выдать запросы `Get Message` (Получить сообщение) с тем того, чтобы получить сообщения из соответствующего сектора форума, т. е. в `Get Message` (Получить сообщение) использу-

ется базовый фрейм `Get With ID and Slot` (Получить с идентификатором и слотом); сервер отвечает фреймами `Message` (Сообщение), в которых содержится тема сообщения и само сообщение, указывается шаг, на котором было создано сообщение, и ссылки на любые другие субъекты, указанные в сообщении. В дополнение к обычному набору элементов, которые есть в протоколе `Thousand Parsec` (игроки, предметы и т.п.), также имеются некоторые специальные ссылки, в том числе приоритет сообщения, действия игрока и статус распоряжения. Естественно, клиентская программа также может с помощью фрейма `Post Message` (Послать сообщение), который является транспортом для фрейма `Message` (Сообщение), отправлять сообщения, а также удалять эти сообщения с помощью фрейма `Remove Message` (Удалить сообщение), базирующегося на фрейме `GetMessage` (Получить сообщение).

21.2.3. Объекты, распоряжения и ресурсы

Основная часть процесса взаимодействия со Вселенной осуществляется через серию фреймов, с помощью которых реализовываются функции объектов, распоряжений и ресурсов.

Физическое состояние Вселенной или, по крайней мере той ее части, которой игрок управляет или может видеть, необходимо получать через соединение, т. е. с помощью клиентской программы. Клиентская программа обычно отправляет запрос `Get Object IDs` т. е. `Get ID Sequence` (Получить идентификаторы объектов, т. е. Получить последовательность идентификаторов), на который сервер отвечает фреймом `List of Object IDs` (Список идентификаторов объектов). Затем клиентская программа может запросить подробности об отдельных объектах с помощью запросов `Get Object by ID` (Получить объект по идентификатору), ответом на которые являются фреймы `Object` (Объект), в которых содержатся эти подробности, опять же при условии, что их разрешено видеть игроку, такие как их тип, название, размер, положение, скорость, объекты, которые находятся внутри рассматриваемого объекта, допустимые типы распоряжений и текущие распоряжения. В протоколе также предусмотрен запрос `Get Object IDs by Position` (Получить идентификаторы объектов, находящихся в позиции), который позволяет клиентской программе находить все объекты в пределах указанной сферы пространства.

Клиентская программа, следуя обычной схеме получения последовательности элементов, получает набор допустимых распоряжений при помощи запроса `Get Order Description IDs` (Получить идентификаторы описаний распоряжений) и для каждого идентификатора ID, полученного в ответе `List of Order Description IDs` (Список идентификаторов описаний распоряжений), отправки запроса `Get Order Description` (Получить описание распоряжения) и получения ответа `Order Description` (Описание распоряжения). На протяжении всего времени существования протокола были заметно улучшены реализация распоряжений и реализация очередей распоряжений. Первоначально каждый объект имел одну очередь распоряжений. Клиент выдавал запрос `Order` (Распоряжение), содержащий тип распоряжения, целевой объект и другие сведения, получал ответ `Outcome` (Результаты) с подробным изложением ожидаемых результатов исполнения распоряжения и после того, как распоряжение было выполнено, получал фрейм `Result` (Результат), содержащий фактический результат.

Во второй версии во фрейм `Order` (Распоряжение) было добавлено содержимое фрейма `Outcome` (Результаты), так как, поскольку он базируется на описании распоряжения, то для него не требуется обращаться на сервер, а фрейм `Result` (Результат) был полностью удален. В последней версии протокола очередь распоряжений к объектам была реструктурирована и были добавлены фреймы `Get Order Queue IDs` (Получить идентификаторы очередей распоряжений), `List of Order Queue IDs` (Список идентификаторов очередей распоряжений), `Get Order Queue` (Получить очередь с распоряжениями) и `Order Queue` (Очередь с распоряжениями), которые работают аналогично функциям обработки сообщений и форумов [2]. Фреймы `Get Order` (Получить распоряжение) и `Remove Order` (Удалить распоряжение), для каждого из которых также требуется запрос `Get WithID Slot` (Получить со слотом идентификаторов), позволяют клиентской программе получать доступ к очередям и удалять из очереди распоряжения, соответственно. Теперь фрейм `Insert`

`Order` (Добавить распоряжение) выступает в роли транспорта полезной нагрузки `Order` (Распоряжение); это было сделано для того, чтобы позволить другому фрейму, `Probe Order` (Проверить состояние распоряжения), который в некоторых случаях используется клиентской программой, получать информацию, используемую локально.

Описания ресурсов также представляет собой последовательность элементов: запрос `Get Resource Description IDs` (Получить идентификаторы описаний ресурсов), ответ `List of Resource Description IDs` (Список идентификаторов описаний ресурсов) и серия запросов `Get Resource Description` (Получить описание ресурса) и ответов `Resource Description` (Описание ресурса).

21.2.4. Работа с Моделями

Работа с моделями в протоколе `Thousand Parsec` разделена на работу с четырьмя различными подкатегориями: категориями, компонентами, свойствами и моделями.

Категории различаются по типам различных моделей. Двумя наиболее часто используемыми типами моделей являются корабли и вооружение. Создание категории достаточно простое, поскольку категория состоит только из названия и описания; в самом фрейме `Category` (Категория) содержатся только эти две строки. В соответствие с набором каждая категория с помощью запроса `Add Category` (Добавить категорию), который является транспортом для фрейма `Category` (Категория), добавляется на склад моделей `Design Store`. Все остальное управление категориями осуществляется по обычной схеме работы с последовательностью элементов с помощью запроса `Get Category IDs` (Получить идентификаторы категорий) и ответа `List of Category IDs` (Список идентификаторов категорий).

Компоненты представляют собой различные части и модули, которые входят в состав моделей. Они могут быть всем, чем угодно, от корпуса корабля или ракеты и до ракетной установки, на которой находится ракета. Компоненты чуть сложнее, чем категории. Во фрейме `Component` (Компонент) содержится следующая информация:

- Название и описание компонента.
- Список категорий, к которым принадлежит компонент.
- Функция `Requirements` (Требования), написанная на языке компонентов протокола `Thousand Parsec (TPCL)`.
- Список свойств и их значений.

Обратите внимание на функцию `Requirements` (Требования), связанную с компонентом. Поскольку компоненты являются теми частями, их которых собирается корабль, вооружение или другой конструируемый объект, необходимо гарантировать, что они действительно будут именно такими, которые можно добавить в модель. Функция `Requirements` (Требования) проверяет, что каждый компонент, добавляемый к модели, соответствует правилам, определенным в других ранее добавленных компонентах. Например, в наборе правил `Missile and Torpedo Wars` (Ракетные и торпедные войны), нельзя в корабле иметь ракету `Alpha Missile` в случае, если в нем нет ракетной установки `Alpha Missile Tube`. Эта проверка осуществляется как на стороне клиентской программы, так и на стороне сервера, именно поэтому функция должна быть целиком описана во фрейме протокола, и именно поэтому для нее был выбран компактный язык описания (язык `TPCL`, описываемый далее в этой главе).

Обращение ко всем свойствам модели происходит через фрейм `Property` (свойство). В каждом наборе правил задается набор свойств, используемых в игре. Они обычно включают в себя такие вещи, как количество ракетных установок определенного типа, размещаемых на корабле, или тип брони, используемый с определенным типом корпуса корабля. Точно также как и во фреймах

Component (Компонент), во фреймах Property (Свойство) применяется язык TPCL. Во фрейме Property (Свойство) содержится следующая информация:

- Наименование (отображаемое) и описание свойства.
- Список категорий, которые обладают данным свойством.
- Имя (правильный идентификатор языка TPCL) свойства.
- Ранг свойства.
- Функции Calculate (Вычислить) и Requirements (Требования), написанные на языке компонентов протокола Thousand Parsec (TPCL).

Ранг свойства используется для выявления различий в иерархии зависимостей. В языке TPCL, функция может не зависеть от тех свойств, ранг которых меньше или равен рангу данного свойства. Это означает, что если ранг защиты равен 1, а ранг скрытности равен 0, то свойство скрытности не может напрямую зависеть от свойства защиты. Такое ранжирование было реализовано в качестве средства сокращения циклических зависимостей. Функция Calculate (Вычислить), используется для того, чтобы определить, как следует отображать некоторое свойство в зависимости от различий, выявленных при измерении. В наборе правил Missile and Torpedo Wars (Ракетные и торпедные войны) для импорта в игру свойств из файла данных игры использован язык XML. На рис.21.2 показан пример свойства из этой игры.

```
<prop>
<CategoryIDName>Ships</CategoryIDName>
<rank value="0"/>
<name>Colonise</name>
<displayName>Can Colonise Planets</displayName>
<description>Can the ship colonise planets</description>
<tpclDisplayFunction>
    (lambda (design bits) (let ((n (apply + bits))) (cons n (if (= n 1) "Yes" "No")))
  )
</tpclDisplayFunction>
<tpclRequirementsFunction>
    (lambda (design) (cons #t ""))
</tpclRequirementsFunction>
</prop>
```

Рис.21.2: Пример свойств

В этом примере у нас есть свойство, принадлежащее категории кораблей Ships, имеющее ранг 0. Это свойство называется Colonise (Колонизация) и относится к возможности использовать корабли при колонизации планет. Беглый взгляд на функцию Calculate (Вычислить), указанную здесь как tpclDisplayFunction, показывает, что в зависимости от того, есть ли у корабля такое свойство, эта функция выдает в ответ на запрос либо значение «Yes» («Да»), либо значение «No» («Нет»). Такое добавление свойств позволяет разработчику наборов правил более детально управлять метриками игры, легко их сравнить и выводить их в дружественном для игрока виде.

Фактическая модель кораблей, вооружения и другие игровые артефакты создаются и изменяются с помощью фрейма Design (Модель) и связанных с ним других фреймов. При создании кораблей и вооружений обычно используются все компоненты и свойства, имеющиеся в текущем наборе правил. Поскольку в правилах для моделей уже есть TPCL функций Requirements (Требования) для компонентов и свойств, создание модели несколько упрощается. Во фрейме Design (Модель) содержится следующая информация:

- Название и описание модели.
- Список категорий, к которым принадлежит модель.
- Счетчик числа экземпляров данной модели.
- Владелец модели.
- Список идентификаторов компонентов и их количество.

- Список свойств и соответствующая им строка, отображаемая на экране.
- Отзыв о модели.

Этот фрейм немного отличается от других. В частности, поскольку модель — это тот элемент в игре, у которого есть владелец, имеется связь каждой модели с ее владельцем. В модели с помощью счетчика также отслеживается количество экземпляров модели.

21.2.5. Администрирование сервера

Также существует расширение протокола, используемое для администрирования сервера и позволяющее дистанционно управлять работающими серверами поддержки. Стандартный вариант использования представляет собой подключение к серверу клиентской программы администрирования, возможно, предоставляющей интерфейс, напоминающий командную строку или графический интерфейс конфигурационной панели, с помощью которых осуществляется изменение настроек или выполняются другие задачи управления. Но в случае однопользовательских игр можно использовать другие более специализированные средства управления, которые работают в фоновом режиме.

Точно также как и в случае игрового протокола, описанного в предыдущих разделах, клиентская программа администрирования сначала договаривается о подключении (через порт, отличающийся от обычного игрового порта) и с помощью запросов `Connect` (Подключение) и `Login` (Регистрация) происходит проверки подлинности. После подключения, клиентская программа может получать журнальные сообщения и отправлять команды на сервер.

Журнальные сообщения передаются клиентской программе через фреймы `Log Message` (Журнальное сообщение). В них указывается уровень серьезности сообщения и его текст; в зависимости от соответствующего контекста, в клиентской программе можно выбирать отображать все получаемые журнальные сообщения, некоторые из них или вообще их не отображать.

Сервер также может отправить фрейм `Command Update` (Обновить команду), указывающий клиенту, что нужно заполнить или обновить его локальный набор команд; в ответ на фрейм `Get Command Description IDs` (Получить идентификаторы описания команд) сервер перенаправит клиентской программе список поддерживаемых команд. Затем можно получить описания каждой отдельной команды, отправив для каждой команды фрейм `Get Command Description` (Получить описание команды), на который сервер ответит с помощью фрейма `Command Description` (Описание команды).

Такой обмен данными функционально очень похож (и, по сути, изначально на нем основывался) на обмен, используемый фреймами распоряжений в основном игровом протоколе. Он позволяет в некоторой степени локально проверять команды, которые набирает пользователь, что ведет к минимизации использования сети. Протокол администрирования разрабатывался уже тогда, когда игровой протокол был уже достаточно зрелым; чтобы не начинать все с нуля, разработчики взяли из игрового протокола уже существующие функциональные возможности и в те же самые библиотеки протоколов добавили код.

21.3. Поддерживаемые функциональные возможности

21.3.1. Хранение данных на сервере

В играх *Thousand Parsec* точно также как и во многих играх пошаговых стратегий, есть возможность продолжать игру в течение достаточно долгого времени. Поскольку часто игра продолжается гораздо дольше суточных ритмов отдельных игроков, серверный процесс в течение этого длительного периода может быть по ряду причин досрочно остановлен. Чтобы разрешить игрокам возобновлять игру с того места, где они остановились, на серверах *Thousand Parsec* есть возможность

сохранять состояние всей Вселенной (или даже нескольких Вселенных) в базе данных. Эти функциональные возможности также касаются сохранения состояния однопользовательских игр, которые более подробно будут рассмотрены далее в этом разделе.

На основном сервере `tpserver-cpp` предоставляется абстрактный интерфейс хранения данных и модульная система плагинов для подключения различных баз данных в качестве хранилища. На момент написания данной главы сервер `tpserver-cpp` поставлялся с модулями для MySQL и SQLite.

В абстрактном классе `Persistence` описаны функции, позволяющие серверу сохранять, обновлять или осуществлять поиск различных элементов игры (так, как это описано в разделе «Анатомия звездной империи Star Empire»). Когда в различных местах кода сервера происходит изменение состояния игры, база данных обновляется, так что не важно, где произойдет остановка сервера или его сбой; когда сервер будет запущен снова с сохраненными данными, игра продолжится с того же самого запомненного места.

21.3.2. Язык компонентов Thousand Parsec Component Language

Существует язык описания компонентов Thousand Parsec Component Language (TPCL), который позволяет клиентским программам создавать модели локально без взаимодействия с сервером — в результате обеспечивается мгновенная обратная реакция на изменение свойств, внешнего вида и правильности создания моделей. Это позволяет игроку интерактивно создавать, например, новые классы космических кораблей, изменения в соответствие с имеющейся технологией конструкцию корабля, двигатели, приборы, защиту, вооружение и многое другое.

Язык TPCL является подмножеством языка Scheme с небольшими изменениями, но достаточно близкими к стандарту языка Scheme R5RS, так что можно использовать любой совместимый интерпретатор. Язык Scheme первоначально был выбран из-за его простоты, из-за большого количества прецедентов его использования в качестве встроенного языка, наличия интерпретаторов, реализованных на многих других языках, и, самое главное, из-за того, что это проект с открытым кодом и огромным количеством документации как по его использованию, так и по разработке для него интерпретаторов.

Рассмотрим следующий пример функции `Requirements` (Требования), написанной на языке TPCL и используемой компонентами и свойствами. Функция входит в набор правил, размещаемых на серверной стороне и передаваемых клиентской программе по игровому протоколу:

```
(lambda (design)
  (if (> (designType.MaxValue design) (designType.Size design))
      (if (= (designType.num-hulls design) 1)
          (cons #t "")
          (cons #f "Ship can only have one hull"))
      )
    (cons #f "This many components can't fit into this Hull")
  )
)
```

Читатели, знакомые с языком Scheme, без сомнения, легко разберутся с этим кодом. В игре (в клиентской программе и на сервере) этот код используется для проверки свойств других компонентов (`MaxSize`, `Size` и `Num-Hulls`) с тем, чтобы убедиться, что конкретный компонент может быть добавлен в модель. Сначала проверяется, что `Size` (Размер) компонента не превышает максимального размера, допустимого в модели, затем проверяется, что в модели не используются какие-нибудь другие корпуса, что гарантирует, что в конструкции нет никаких других корпусов. В последней проверке нам сообщается, что эта функция `Requirements` (Требования) зависит от выбираемого корпуса корабля.

21.3.3. Описание сражения - BattleXML

На войне имеет значение каждый бой, от короткой перестрелки в глубоком космосе между эскадрильями малых слабо вооруженных кораблей-разведчиков и до масштабного окончательного столкновения двух флагманских флотов в небе над столицей мира. Во фреймворке Thousand Parsec конкретные особенности боевых действий обрабатываются в соответствие с набором правил и функциональные возможности клиентской программы никак не могут повлиять на детали боевых действий - как правило, игрок будет проинформирован о начале и результатах боевых действий с помощью сообщений и произойдут соответствующие изменения объектов (например, удаление уничтоженных кораблей). Хотя игрок, обычно, сосредотачивает свое внимание на общем ходе боевых действий, которое происходит в соответствии с наборами правил сложной механики боя, он может иметь право (или, по крайней мере, удовольствие) изучить детали боя более подробно.

Все это поступает в виде данных BattleXML. Данные о боевых действиях состоят из двух основных частей: медийного описания, в котором указываются подробности используемого графического сопровождения, и определения сражения, в котором указывается то, что, на самом деле, произошло во время битвы. Все это предназначено для тех, кто наблюдает за сражением и может в настоящее время пользоваться двумя вариантами графики: 2D и 3D. Конечно, поскольку характер битв полностью определен в наборе правил, то код, обрабатывающий набор правил, обязан обрабатывать данные BattleXML.

Медийное описание связано с природой наблюдателя и хранится в каталоге или архиве, содержащем XML-данных и все графические файлы или файлы модели, на которые есть ссылки. В самих данных описывает то, как средства мультимедиа должны использоваться для каждого типа корабля (или другого объекта), их анимация для таких действий, как стрельба и уничтожение, а также мультимедийная информация и подробности их вооружения. Предполагается, что файлы размещаются рядом с самим файлом XML и ссылки на родительские каталоги не допускаются.

Определение сражения не зависит от наблюдателя и мультимедийных средств. Во-первых, в нем описывается ряд объектов, имеющихся с каждой стороны в начале битвы, и указываются уникальные идентификаторы и информация, такая как имя, описание и тип. Затем, описывается каждый раунд битвы: движение объектов, огонь вооружений (с указанием объекта, ведущего огонь, и цели), повреждения объектов, уничтожение объектов и журнальное сообщение. Насколько подробно описывается каждый раунд сражения, диктуется набором правил.

21.3.4. Метасервер

Поиск общедоступного сервера Thousand Parsec, на котором можно играть, очень похож на определение местоположения одинокого невидимого корабля-разведчика в глубоком космосе - пугающая перспектива, если не знать, где искать. К счастью, общедоступные сервера могут оставлять объявления о себе на метасервере, положение которого, как центрального узла, в идеале должно быть хорошо известно игрокам.

В текущей реализации есть метасервер `metaserver-lite`, PHP скрипт, который размещается на некотором центральном месте, например, на сайте Thousand Parsec. Поддерживающие сервера отправляют запрос HTTP, сообщающий об обновлении действий и содержащий тип, местоположение (протокол, хост и порт), набор правил, количество игроков, количество объектов, имя администратора и другую дополнительную информацию. По истечению определенного времени (по умолчанию, через 10 минут) листинги на сервере становятся неактуальными, поэтому предполагается, что серверы периодически обновляют информацию на метасервере

Тогда скрипт может, когда происходит вызов без указания действия, выдать список серверов с подробной информацией, имеющейся на сайте, и со вставленными ссылками с адресами URL (как правило, с именем схемы `tp://`), по которым можно щелкнуть мышкой. Действие `badge` позволяет, в

качестве альтернативного варианта, предоставлять листинги, имеющиеся на сервере, в компактном формате "бейджик".

Чтобы получить список имеющихся серверов, клиентские программы могут с помощью действия `get` сделать запрос к метасерверу. В этом случае, метасервер возвращает клиентской программе для каждого сервера, указанного в списке, один или несколько фреймов `Game` (Игра). В `tpclient-rwux` результирующий список выдается в браузере сервера в окне первоначального подключения.

21.3.5. Однопользовательский режим

Для того, чтобы была возможность поддерживать сетевые многопользовательские игры, фреймворк `Thousand Parsec` разрабатывался с нуля. Тем не менее, ничто не мешает игроку вести стрельбу на локальном сервере, подключать несколько интеллектуальных клиентских программ и совершать гиперпрыжки в собственной однопользовательской Вселенной, которую можно завоевать. В проекте определены некоторые стандартные метаданные и функции, обеспечивающие упорядочивание этого процесса, что упрощает установку до запуска визарда в графическом пользовательском интерфейсе или до двукратного щелчка по файлу со сценарием.

Ядром таких функциональных возможностей является XML DTD, определяющий формат метаданных, относящихся к возможностям и свойствам каждого компонента (например, сервера, интеллектуальной клиентской программы, набора правил). Пакеты компонентов поставляются с одним или несколькими такими файлами XML, и, в конечном итоге, все эти метаданные объединяются в ассоциативный массив, состоящий из двух основных частей: серверов и интеллектуальных клиентских программ. В метаданных сервера, как правило, можно найти метаданные для одного или нескольких наборов правил - они помещаются сюда потому, что, набор правил может быть реализован для более чем одного сервера, некоторые детали конфигурирования могут отличаться, так что в целом для каждой реализации необходимы отдельные метаданные. В каждой записи каждого из этих компонентов содержится следующая информация:

- Данные с описанием, включающие в себя краткое имя (двоичное), длинное имя (описательное) и описание.
- Установленную версию компонента и более раннюю версию, сохраненные данных для которой совместимы с установленной версией.
- Параметры командной строки (если они используются) и любые другие параметры, передаваемые в компонент принудительно.
- Набор параметров, которые может указывать игрок.

Параметры, устанавливаемые принудительно, игроком не конфигурируются и это, как правило, те параметры, которые позволяют компонентам надлежащим образом функционировать в локальном однопользовательском контексте. Параметры, конфигурируемые игроком, имеют свой собственный формат, в котором указываются такие подробности, как название и описание, тип данных, значения, используемые по умолчанию, и диапазон значений, а также формат строки для добавления параметра в основную командную строку.

Хотя возможны специальные случаи (например, предустановленные игровые конфигурации для конкретных клиентских программ), типичный процесс построения однопользовательской игры включает в себя выбор набора совместимых компонентов. Выбор клиентской программы происходит неявно, поскольку игрок уже запустил одну из них с тем, чтобы сыграть в игру; хорошо разработанная клиентская программа помогает пользователю настроить все остальное. Естественно, следующее, что нужно сделать, это выбрать набор правил, так что игроку предоставляется список - на данный момент, ему нет необходимости вникать в детали, связанные с сервером. В случае, если выбранный набор правил реализован на нескольких установленных серверах (возможная, но редкая ситуация), игроку предлагается выбрать один из них; в противном случае соответствующий сервер выбирается автоматически. Далее, игроку будет предложено сделать настройки параметров для набора правил и для сервера, для которых уже указанные приемлемые используемые

по умолчанию значения, которые взяты из метаданных. И наконец, если установлены какие-либо совместимые интеллектуальные клиентские программы, то игроку будет предложено настроить одну из них или несколько с тем, чтобы против них играть.

После того, как игра сконфигурирована, клиентская программа, используя информацию о командной строке, полученную из метаданных, запускает локальный сервер с соответствующими параметрами конфигурации (в том числе набором правил, его параметрами и любыми параметрами, которые добавляются к конфигурации сервера). Как только будет получено подтверждение, что сервер работает и к нему есть доступ, может быть, аналогичным образом с помощью расширений административного протокола, который был описан выше, будет запущена каждая из указанных интеллектуальных клиентских программ и будет проверено, что все они успешно подключены к игре. Если все пойдет хорошо, то тогда клиентская программа подключится к серверу точно также, как если бы она подключалась к онлайн-игре, и игрок может начать исследовать, торговать, завоевывать и делать что-нибудь еще, что есть в универсуме других возможностей.

Альтернативой, причем очень важной, использования функциональных возможностей однопользовательского режима является сохранение и загрузка игр и, что более или менее эквивалентно, загрузка сценариев, готовых для игры. В этом случае в сохраняемых данных (возможно, хотя и не обязательно, в одном файле) запоминаются конфигурационные данные однопользовательской игры, а также данные о текущем состоянии самой игры. Если в системе игрока установлены все необходимые компоненты совместимых версий, то запуск сохраненной игры или сценария выполняется абсолютно автоматически. Для сценариев, в частности, таким образом реализована привлекательная возможность вступить в игру одним щелчком мыши. Хотя во фреймворке Thousand Parsec в настоящее время нет специального редактора сценариев или клиентской программы с режимом редактирования, в концепции фреймворка, кроме обычного использования набора правил, предусмотрены некоторые средства сохранения данных и проверки их непротиворечивости и совместимости.

До сих пор описание этой функциональной возможности было сравнительно абстрактным. Если рассматривать ее более конкретно, то в проекте Thousand Parsec только клиентская библиотека Python `libtpclient.py` имеет в настоящее время полную реализацию однопользовательского механизма. В библиотеке предоставлен класс `SinglePlayerGame`, который при создании собственных экземпляров автоматически агрегирует все метаданные однопользовательского режима, доступные в системе (естественно, есть определенные рекомендации, касающиеся того, какие файлы XML должны быть установлены на данной платформе). После этого к объекту можно будет обращаться для получения информации по имеющимся компонентам: серверам, наборам правил, интеллектуальным клиентским программам и параметрам, которые хранятся в виде словарей (ассоциативные массивы языка Python). После того, как будет завершен общий процесс создания игры, описанный выше, типичная клиентская программа сможет выполнять следующие действия:

1. С помощью `SinglePlayerGame.rulesets` сделать запрос списка доступных наборов правил и с помощью `SinglePlayerGame.rname` настроить объект в соответствие с выбранным набором правил.
2. С помощью `SinglePlayerGame.list_servers_with_ruleset` сделать запрос списка серверов, на которых реализован данный набор правил, если необходимо, предложить пользователю выбрать один из них и (или только) сконфигурировать набор правил с помощью метода `SinglePlayerGame.sname`.
3. С помощью `SinglePlayerGame.list_rparams` и `SinglePlayerGame.list_sparams` получить соответственно набор параметров для сервера и набора правил, и предложить игроку их сконфигурировать.
4. С помощью `SinglePlayerGame.list_aiclients_with_ruleset` найти имеющиеся интеллектуальные клиентские программы и предложить игроку сконфигурировать одну или нескольких из них с помощью параметров, которые можно получить через `SinglePlayerGame.list_aiparams`.

5. Запустить игру, вызвав метод `SinglePlayerGame.start`, который в случае, если подключение будет успешным, вернет порт TCP/IP.
6. В конце концов, вызвав метод `SinglePlayerGame.stop`, закончить игру (и уничтожить все процессы, запущенные сервером и интеллектуальными клиентскими программами).

Флагманская клиентская программа проекта Thousand Parsec, `tpclient-puyx`, представляет собой дружественный пользователю визард, который, следуя такой процедуре, сначала предлагает вызвать для загрузки сохраненную игру или файл сценария. Дружественная для пользователя последовательность действий, разработанная для этого визарда, является примером хорошей разработки, созданной на основе процесса разработки программ с открытым исходным кодом, используемого в проекте: разработчик первоначально предложил совсем другой процесс, больше напоминающий работу автомобиля, когда все спрятано под капотом, но обсуждения в сообществе и некоторые совместные разработки привели к результату, более удобному для игрока.

И, в заключение, сохранение игр и сценариев в настоящее время реализовано на практике в сервере `tpserver-cpp`, причем с поддержкой функциональных возможностей `libtpclient-pu` и интерфейса `tpclient-puyx`. Это достигнуто с помощью модуля, сохраняющего данные и использующего SQLite — общедоступную СУБД с открытым исходным кодом, для которой не нужен внешний процесс, а все базы данных хранятся в одном файле. Сервер с помощью специально заданных параметров конфигурируется так, чтобы использовался модуль, сохраняющий данные в SQLite, если таковой имеется, и, как и обычно, на протяжении всей игры постоянно обновлялся файл базы данных (размещаемый в месте хранения временных файлов). Когда игрок решит сохранить состояние игры, файл базы данных копируется в заданное место, и к нему добавляется специальная таблица, в которой хранятся конфигурационные данные отдельного игрока. Читателю должно быть понять, как их впоследствии загружать.

21.4. Усвоенные уроки

Создание и разработка обширного фреймворка Thousand Parsec предоставило разработчикам достаточно много возможностей, позволяющих оглянуться назад и оценить проектные решения, которые были сделаны на этом пути. Разработчики исходного ядра Тим Ансел (Tim Ansell) и Ли Бегг (Lee Begg), создавшие исходный фреймворк с нуля, поделились с нами некоторыми рекомендациями, касающимися запуска аналогичного проекта.

21.4.1. Что работает

Основным ключевым аспектом в разработке проекта Thousand Parsec было решение выделить и создать подмножество фреймворка, за которым бы последовала реализация. Этот итеративный и поэтапный процесс разработки позволил фреймворку расти органически, во время которого новые функции добавлялись без возникновения проблем. Это привело напрямую к решению разрабатывать версии протокола Thousand Parsec, что привело к ряду крупных успехов. Использование версий протокола позволило фреймворку расти непрерывно, причем, при этом, в него по ходу дела добавлялись игры, использующие новые методики.

Когда разрабатывается такой обширный фреймворк, важно следовать подходу с очень коротким циклом постановки целей и их реализации. Короткие итерации, порядка недели - для второстепенных релизов, позволило проекту быстро двигаться вперед с немедленными откатами назад на этом пути. Другим успешным аспектом реализации была клиент-серверная модель, которая позволила разрабатывать клиентские программы отдельно от логики игры. Отделение логики игры от клиентского программного обеспечения оказалось важным для общего успеха проекта Thousand Parsec.

21.4.2. Что не работает

Основным недостатком фреймворка Thousand Parsec оказалось решение использовать двоичный протокол. Как вы можете себе представить, отладка двоичного протокола оказалась совсем не веселой задачей, и это привело к затягиванию многих сеансов отладки. Мы настоятельно бы рекомендовали, чтобы никто в будущем не вставал на этот путь. При разработке протокола также следовало бы проявлять больше гибкости; когда создается протокол, важно реализовывать только базовые возможности, которые необходимы.

Наши итерации, время от времени, становились слишком долгими. Когда управление таким большим фреймворком осуществляется по принципам разработки проектов с открытым исходным кодом, то для того, чтобы разработка шла гладко, важно на каждой итерации добавлять небольшое количество функций.

21.4.3. Заключение

Точно так же, как и при инспекции остова корпуса тяжелого боевого корабля на орбитальной верфи, мы прошлись по различным особенностям архитектуры проекта Thousand Parsec. Хотя общие проектные критерии гибкости и расширяемости сидели в умах разработчиков с самого начала, для нас, когда мы рассматриваем историю разработки фреймворка, очевидно, что только экосистема с открытыми исходными кодами, изобилующая свежими идеями и точками зрения, была в состоянии предоставить новый пласт возможностей, сохраняя при этом функционирование и целостность проекта. Это особенно амбициозный проект, и, как и во многих других подобных проектах из зоны существования открытых исходных кодов, в нем еще многое предстоит сделать; это наша надежда и ожидание того, что с течением времени проект Thousand Parsec будет продолжать развиваться и расширять свои возможности и для него будут разрабатываться все новые и все более сложные игры. В конце концов, путешествие в тысячу парсеков начинается с одного шага.

Примечания

1. К числу некоторых отличных коммерческих примеров, получивших вдохновение от Thousand Parsec, относятся *VGA Planets* и *Stars!*, а также серии игр *Master of Orion*, *Galactic Civilizations* и *Space Empires*. Для тех, кто не знаком с этими названиями, серии игр *Civilization* — популярный пример игр того же стиля, но в другой обстановке. Также есть ряд игр реального времени в жанре 4X, например, *Imperium Galactica* и *Sins of a Solar Empire*.
2. На самом деле, наоборот: сообщения и форумы были созданы на основе распоряжений из второй версии протокола.

22. Фреймворк Violet

Глава 22 из книги "[Архитектура приложений с открытым исходным кодом](#)", том 1.

В 2002 году я написал для студентов учебник по объектно-ориентированному проектированию и использованию шаблонов [Hor05]. Как и в случае многих книг, его появление было вызвано разочарованием, связанным с канонической учебной программой. Часто студенты информатики обучаются проектированию классов на первом курсе программирования, а затем у них отсутствует всякая дальнейшая практика объектно-ориентированного проектирования вплоть до курса инженерии программного обеспечения более высокого уровня. В этом курсе студенты спешно в течение пары недель проходят UML и шаблоны проектирования, что дает не более чем иллюзию знания. Моя книга была написана в поддержку односеместрового курса для студентов, имеющих опыт программирования на языке Java и умеющих использовать основные структуры данных (обычно курсы CS1/CS2 на базе Java). В книге рассматриваются принципы объектно-ориентированного проектирования и использование шаблонов проектирования в контексте знакомых ситуаций. Например, шаблон проектирования Decorator вводится вместе с классом

JScrollPane на Swing в надежде на то, что этот пример будет более запоминающимся, чем канонический пример потоков Java.

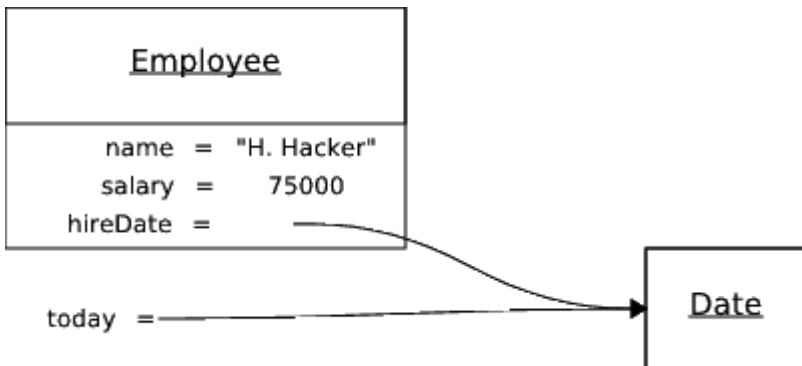


Рис.22.1: Диаграмма объектов в Violet

Мне для книги нужно было упрощенное подмножество языка UML: диаграммы классов, диаграммы последовательностей и вариант диаграмм объектов Java, в которых указываются ссылки на объект (рис. 22.1). Я также хотел, чтобы студенты рисовали свои собственные диаграммы. Тем не менее, коммерческие варианты, например, Rational Rose, были не только дороги, но и громоздки для изучения и использования [Shu05], а альтернативные варианты с открытым исходным кодом, в которых диаграммы задавались с помощью текстовых объявлений, а не обычным щелчком мыши, которые были доступны в то время, были слишком ограниченными или имели ошибки, не позволяющие их использовать. В частности, в ArgoUML были серьезные проблемы с диаграммами последовательностей.

Я решил попробовать свои силы в реализации простейшего редактора, который (а) будет полезен студентам и (б) будет примером фреймворка, с которым студенты смогут разобраться и смогут его модифицировать. Так родился редактор Violet.

22.1. Введение в Violet

Violet является легковесным редактором языка UML, предназначена для студентов, преподавателей и авторов, которым нужно быстро создавать простые диаграммы UML. Он очень прост в освоении и использовании. Он рисует диаграммы классов, последовательностей, состояний, объектов и сценариев использования (use-case). (С тех времен были добавлены другие типы диаграмм). Это кроссплатформенное программное обеспечение с открытым исходным кодом. В качестве своего ядра Violet использует простой, но гибкий фреймворк работы с графикой, который позволяет в полной мере использовать возможности графики Java 2D API.

Пользовательский интерфейс Violet преднамеренно простой. Вам для того, чтобы вводить атрибуты и методы, не придется проходить через утомительную последовательность диалогов. Вместо этого, вы просто набираете их в текстовом поле. С помощью нескольких щелчков мыши, вы можете быстро создавать привлекательные и полезные диаграммы.

Violet не пытается стать программой для использования UML промышленного уровня. Вот некоторые возможности, отсутствующие в Violet:

- Violet не генерирует исходный код из диаграмм UML или диаграммы UML из исходного кода.
- Violet не осуществляет никакой семантической проверки моделей; вы можете использовать Violet для рисования противоречивых диаграмм.
- Violet не создает файлы, которые могут быть импортированы в другие инструментальные средства, работающие с UML, и не может читать файлы моделей других инструментальных средств.

- Violet не пытается автоматически выполнять компоновку диаграмм, за исключением простой возможности "привязки к сетке".

(Попытка решить некоторые из этих ограничений позволило создать хорошие студенческие проекты).

Когда Violet создал культ дизайнеров, которые хотели чего-то большего, чем просто набросок на салфетке, но менее сложного, чем инструментальные средства UML промышленного уровня, я опубликовал код в SourceForge под лицензией GNU General Public License. Начиная с 2005 года к проекту присоединился Александр Пелегрин (Alexandre de Pellegrin), предложивший плагин для Eclipse и более красивый пользовательский интерфейс. С тех пор он сделал в архитектуре множество изменений и в настоящее время он является основным разработчиком, сопровождающим проект (primary maintainer).

В этой статье я рассмотрю некоторые из исходных архитектурных решений, выбранных в Violet, а также покажу их эволюцию. Часть статьи сосредоточена на вопросах редактирования графов, но и другие вопросы, например, использование свойств JavaBeans и хранение результатов, архитектура Java WebStart и плагины, должны быть интересны для всех.

22.2. Графический фреймворк

Violet базируется на универсальном фреймворке редактирования графов, который может выдавать изображения и позволяет редактировать узлы и ребра изображений произвольной формы. Редактор Violet для UML использует узлы для отображения классов, объектов, границ активации (в диаграммах последовательностей), и так далее, а ребра — для различных дуг в диаграммах UML. Другой экземпляр графического фреймворка может отображать диаграммы «сущность - отношение» или синтаксические диаграммы.

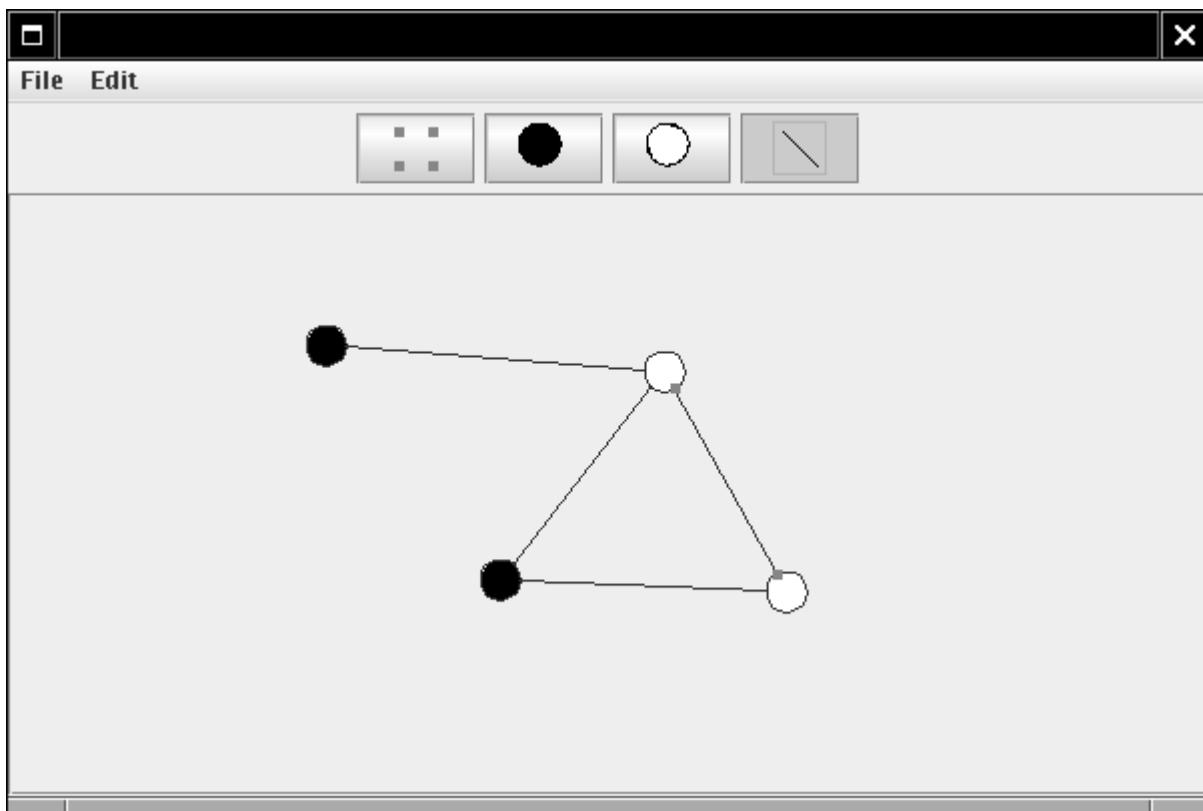


Рис.22.2: Простой экземпляр фреймворка редактирования

Для того чтобы проиллюстрировать фреймворк, рассмотрим редактор для очень простых графов с черно-белыми круглыми узлами и прямыми ребрами (рис.22.2). В классе SimpleGraph определя-

ются прототипные объекты для типов узлов и ребер, иллюстрирующие шаблон прототипирования prototype:

```
public class SimpleGraph extends AbstractGraph
{
    public Node[] getNodePrototypes()
    {
        return new Node[]
        {
            new CircleNode(Color.BLACK),
            new CircleNode(Color.WHITE)
        };
    }
    public Edge[] getEdgePrototypes()
    {
        return new Edge[]
        {
            new LineEdge()
        };
    }
}
```

Прототипные объекты используются для рисования кнопок узлов и ребер в верхней части рисунка 22.2. Они копируются всякий раз, когда пользователь добавляет в граф новый экземпляр узла или ребра. Узел `node` и ребро `Edge` являются интерфейсами со следующими ключевыми методами:

- В обоих интерфейсах есть метод `getShape`, который возвращает объект `Java2D Shape`, представляющий собой узел или ребро.
- The интерфейс `Edge` имеет методы, с помощью которых можно получить начальный и конечный узел ребра.
- Метод `getConnectionPoint` в интерфейсе типа `Node` вычисляет оптимальные точки подсоединения к границе узла (смотрите рис.22.3).
- Метод `getConnectionPoints` интерфейса `Edge` позволяет получать две конечные точки ребра. Этот метод необходим, чтобы рисовать «граббера», которыми помечается текущее выбранное ребро.
- Узел может иметь потомков, которые перемещаются вместе с родителем. Предлагается ряд методов для нумерации и управления потомками.

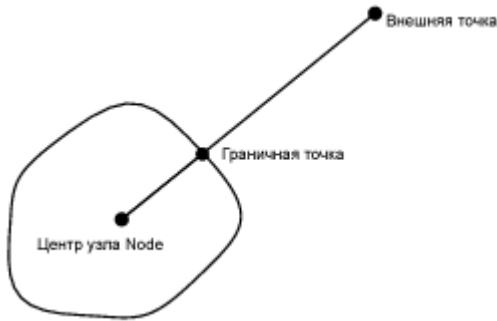


Рис.22.3: Поиск точки присоединения на границе формы узла Node

Удобные классы `AbstractNode` и `AbstractEdge` реализуют ряд таких методов, а классы `RectangularNode` и `SegmentedLineEdge` обеспечивают всю полную реализацию прямоугольных узлов со строкой-заголовком и ребрами, которые состоят из отрезков прямой.

В случае нашего простого редактора графов нам нужны подклассы `CircleNode` и `LineEdge`, в которых есть метод `draw`, метод `contains` и метод `getConnectionPoint`, в которых описывается форма границы узла. Ниже приведен код, а на рис 22.4 показана диаграмма классов для этих классов (нарисованных, конечно, с помощью Violet).

```

public class CircleNode extends AbstractNode
{
    public CircleNode(Color aColor)
    {
        size = DEFAULT_SIZE;
        x = 0;
        y = 0;
        color = aColor;
    }

    public void draw(Graphics2D g2)
    {
        Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
        Color oldColor = g2.getColor();
        g2.setColor(color);
        g2.fill(circle);
        g2.setColor(oldColor);
        g2.draw(circle);
    }

    public boolean contains(Point2D p)
    {
        Ellipse2D circle = new Ellipse2D.Double(x, y, size, size);
        return circle.contains(p);
    }

    public Point2D getConnectionPoint(Point2D other)
    {
        double centerX = x + size / 2;
        double centerY = y + size / 2;
        double dx = other.getX() - centerX;
        double dy = other.getY() - centerY;
        double distance = Math.sqrt(dx * dx + dy * dy);
        if (distance == 0) return other;
        else return new Point2D.Double(
            centerX + dx * (size / 2) / distance,
            centerY + dy * (size / 2) / distance);
    }

    private double x, y, size, color;
    private static final int DEFAULT_SIZE = 20;
}

public class LineEdge extends AbstractEdge
{
    public void draw(Graphics2D g2)
    { g2.draw(getConnectionPoints()); }

    public boolean contains(Point2D aPoint)
    {
        final double MAX_DIST = 2;
        return getConnectionPoints().ptSegDist(aPoint) < MAX_DIST;
    }
}

```

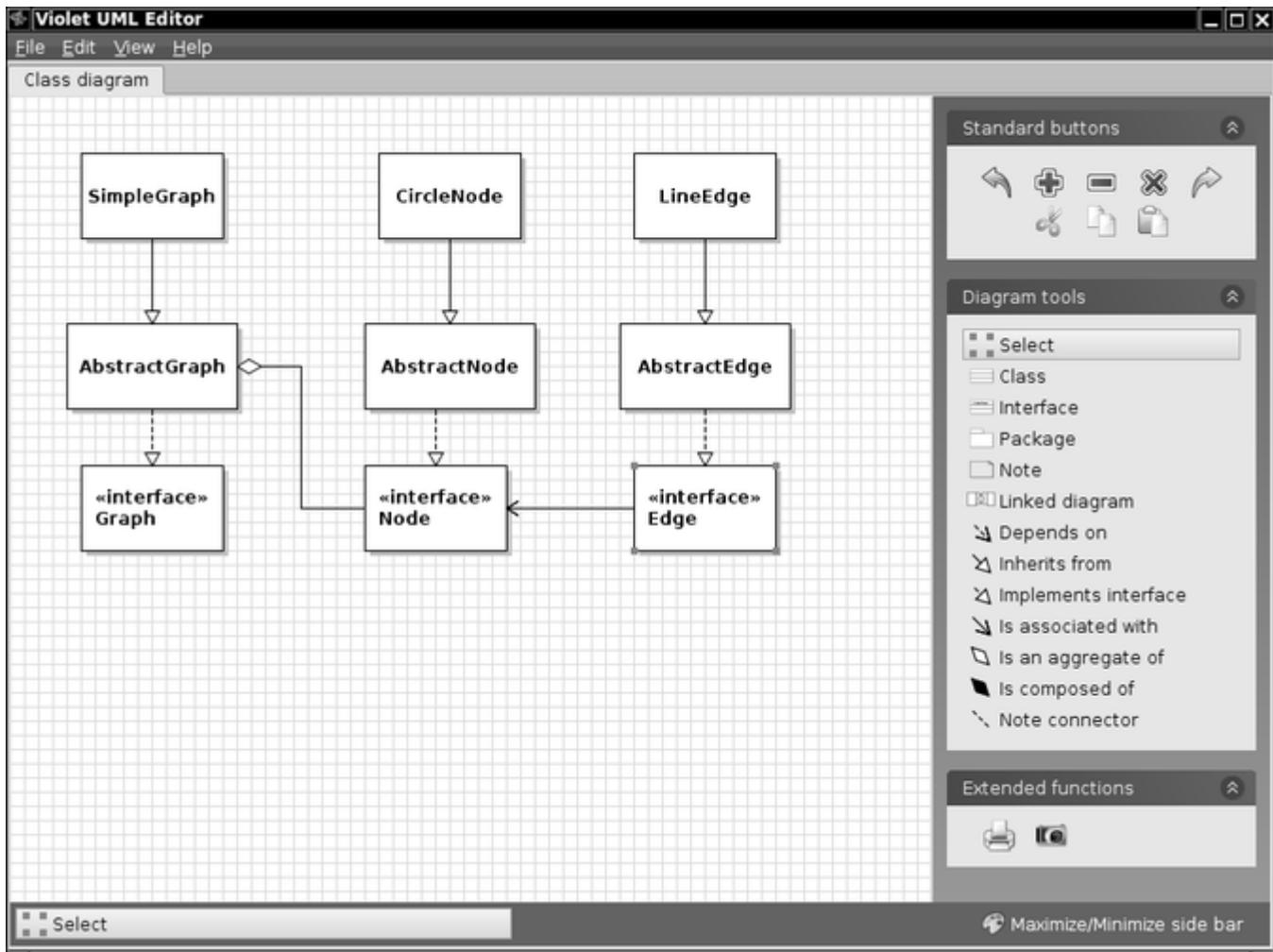


Рис.22.4: Диаграмма классов для простого графа

В целом, Violet предоставляет простой фреймворк для создания редакторов графов. Чтобы получить экземпляр редактора, определяются классы узлов и ребер и предоставляются методы в классе графа, с помощью которых можно получить прототипы объектов узел и ребро.

Конечно, есть и другие графовые фреймворки, например, JGraph [Ald02] и JUNG2. Однако эти фреймворки гораздо более сложные и они являются фреймворками для рисования графов, а не для создания приложений, которые рисуют графы.

22.3. Использование свойств JavaBeans

В золотые деньги использования языка Java на клиентской стороне были разработаны спецификации JavaBeans с целью обеспечить портируемые механизмы для редактирования компонент графического интерфейса GUI в визуальной среде сборки приложений. Предполагалось, что сторонний компонент GUI может быть помещен в любую среду сборки GUI, где его свойства могут быть настроены таким же самым образом, как и стандартные кнопки, текстовые компоненты, и так далее.

В языке Java нет нативных свойств. Вместо этого, могут использоваться свойства JavaBeans, доступ к которым осуществляется при помощи пар методов `get` и `set` или указываемых в сопутствующих классах `BeanInfo`. Кроме того, для визуального редактирования значения свойства могут быть определены *редакторы свойств*. В JDK даже есть несколько базовых редакторов свойств, например, для типа `java.awt.Color`.

Фреймворк Violet в полной мере использует спецификации JavaBeans. Например, в классе `CircleNode` можно пользоваться свойством «цвет» с помощью следующих двух методов:

```
public void setColor(Color newValue)
public Color getColor()
```

Ничего большего не требуется. Редактор графов теперь может редактировать цвет узла круглых узлов (рис.22.5).

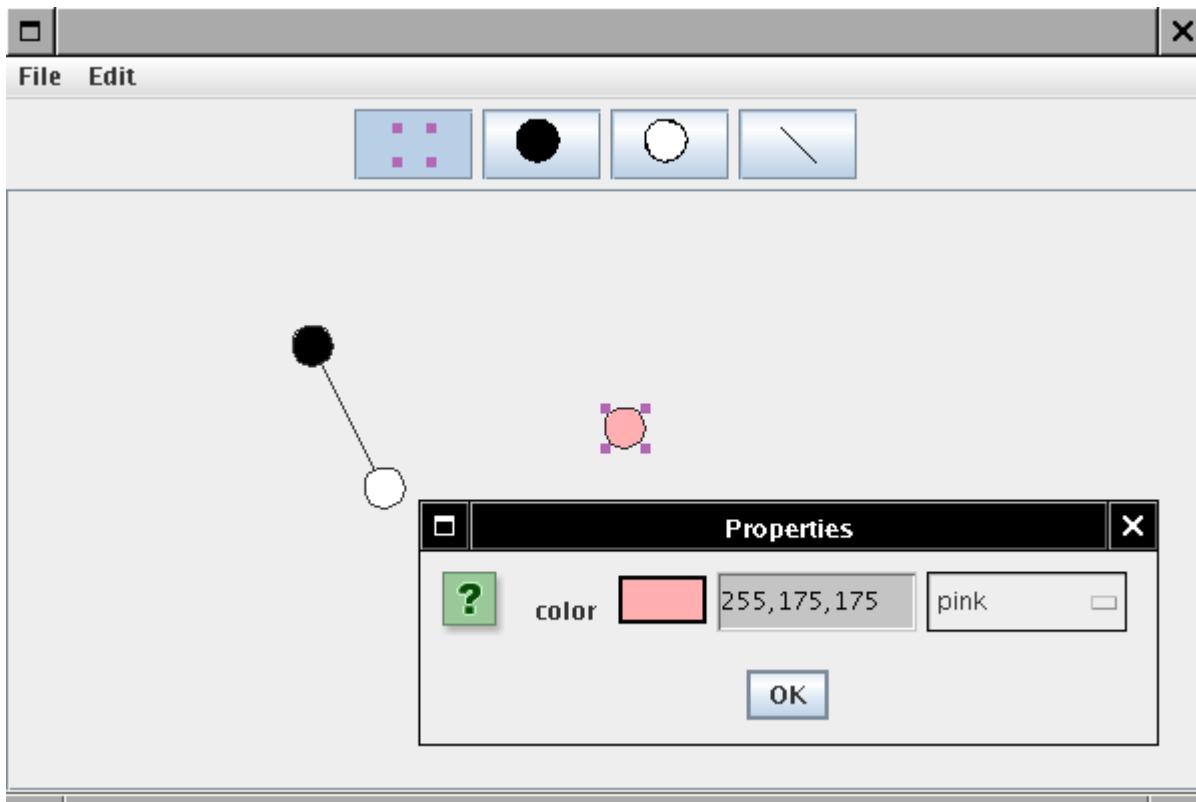


Рис.22.5: Редактирование цвета кружков с помощью редактора цвета, предлагаемого в JavaBeans по умолчанию

22.4. Хранение полученных результатов

Как и любая программа-редактор, Violet должен сохранять в файле творения пользователя и загружать их позже. Я должен был обратиться к спецификации XMI [3], которая была разработана как общий формат обмена для моделей UML. Я посчитал его громоздким, запутанным, и трудным в использовании. Не думаю, что я был одинок - XMI имел репутацию плохо совместимого языка даже для самых простых моделей [PGL+05].

Я решил просто использовать сериализацию языка Java, однако из-за этого было трудно читать старые версии сериализованного объекта, реализация которого с течением времени изменялась. Этую проблему также предвидели архитекторы JavaBeans, которые разработали формат стандарта XML для долгосрочного хранения данных [4]. Объект языка Java, в случае использования Violet, - диаграмма UML, которая сериализуется как последовательность операторов для построения и модификации объекта. Например:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
<object class="com.horstmann.violet.ClassDiagramGraph">
<void method="addNode">
<object id="ClassNode0" class="com.horstmann.violet.ClassNode">
<void property="name">...</void>
```

```

</object>
<object class="java.awt.geom.Point2D$Double">
  <double>200.0</double>
  <double>60.0</double>
</object>
</void>
<void method="addNode">
  <object id="ClassNode1" class="com.horstmann.violet.ClassNode">
    <void property="name">...</void>
  </object>
  <object class="java.awt.geom.Point2D$Double">
    <double>200.0</double>
    <double>210.0</double>
  </object>
</void>
<void method="connect">
  <object class="com.horstmann.violet.ClassRelationshipEdge">
    <void property="endArrowHead">
      <object class="com.horstmann.violet.ArrowHead" field="TRIANGLE"/>
    </void>
  </object>
  <object idref="ClassNode0"/>
  <object idref="ClassNode1"/>
</void>
</object>
</java>

```

Когда класс XMLDecoder читает этот файл, он выполняет эти инструкции (для простоты имена пакетов опущены).

```

ClassDiagramGraph obj1 = new ClassDiagramGraph();
ClassNode ClassNode0 = new ClassNode();
ClassNode0.setName(...);
obj1.addNode(ClassNode0, new Point2D.Double(200, 60));
ClassNode ClassNode1 = new ClassNode();
ClassNode1.setName(...);
obj1.addNode(ClassNode1, new Point2D.Double(200, 60));
ClassRelationShipEdge obj2 = new ClassRelationShipEdge();
obj2.setEndArrowHead(ArrowHead.TRIANGLE);
obj1.connect(obj2, ClassNode0, ClassNode1);

```

До тех пор, пока не изменяется семантика конструкторов, свойств и методов, новая версия программы может прочитать файл, который был подготовлен при помощи старой версии.

Создание таких файлов осуществляется сравнительно просто. Кодировщик автоматически перечисляет свойства каждого объекта и записывает инструкции set для тех значений свойств, которые отличаются от используемых по умолчанию. Большинство базовых типов данных обрабатываются платформой Java; но я должен был предоставить специальные обработчики для Point2D, Line2D и Rectangle2D. Самое главное, что кодировщик должен знать, что граф может быть сериализован как последовательность вызовов методов addNode и connect:

```

encoder.setPersistenceDelegate(Graph.class, new DefaultPersistenceDelegate()
{
  protected void initialize(Class> type, Object oldInstance,
    Object newInstance, Encoder out)
  {
    super.initialize(type, oldInstance, newInstance, out);
    AbstractGraph g = (AbstractGraph) oldInstance;
    for (Node n : g.getNodes())
      out.writeStatement(new Statement(oldInstance, "addNode", new Object[]
      {
        n,
        n.getLocation()
      }));
  }
}

```

```

for (Edge e : g.getEdges())
    out.writeStatement(new Statement(oldInstance, "connect", new Object[]
{
    e, e.getStart(), e.getEnd()
}));
}
);

```

Как только кодировщик был сконфигурирован, сохранение графа стало выглядеть максимально просто:

```
encoder.writeObject(graph);
```

Поскольку декодировщик просто выполняет инструкции, для него настройка не требуется. Графы читаются просто с помощью:

```
Graph graph = (Graph) decoder.readObject();
```

Этот подход работает исключительно хорошо с многочисленными версиями Violet с одним исключением. Последний рефакторинг изменил некоторые имена пакетов и тем самым нарушил обратную совместимость. Одним из вариантов было бы хранить классы в первоначальных пакетах даже если они уже не соответствуют новой структуре пакетов. Вместо этого, разработчик, осуществляющий сейчас сопровождение проекта, предложил использовать трансформер XML для переписывания имен пакетов при чтении устаревших файлов.

22.5. Java WebStart

Java WebStart это технология для запуска приложений из веб-браузера. Сервер, осуществляющий развертывание приложения, отправляет файл JNLP, который запускает вспомогательные приложения в браузере, загружающим и запускающим программу Java. Приложение может иметь цифровую подпись, в этом случае пользователь должен иметь доступ к сертификату, или оно может не иметь подписи и в этом случае программа запускается в изолированной среде, которая имеет немного больше прав, чем песочница апплетов.

Я не думаю, что конечные пользователи могут или должны достоверно разбираться в достоверности цифрового сертификата и его последствиях для безопасности. Одной из сильных сторон платформы Java является ее безопасность, и я понимаю, что важно это использовать.

Песочница Java WebStart является достаточно мощной с тем, чтобы позволить пользователям выполнять полезную работу, в том числе загружать и сохранять файлы, а также пользоваться печатью. Эти операции, с точки зрения пользователя, выполняются надежно и удобно. Пользователь будет предупрежден о том, что приложение хочет получить доступ к локальной файловой системе, а затем он сам выберет файл для чтения или записи. Приложение лишь примет потоковый объект, не имея возможности во время выбора файла заглянуть в файловую систему.

Раздражает то, что когда приложение работает под WebStart, разработчик должен писать специальный код для взаимодействия с FileOpenService и FileSaveService, и еще больше раздражает то, что нет какого-нибудь вызова WebStart API, позволяющего выяснить, было ли приложение запущено с помощью WebStart.

Кроме того, сохранение пользовательских настроек необходимо реализовывать двумя способами: с помощью Java preferences API в случае, когда приложение работает normally, или с помощью сервиса WebStart preference в случае, когда приложение запускается в рамках WebStart. Процесс печати, с другой стороны, полностью прозрачный для прикладного программиста.

С тем, чтобы жизнь программистам сделать существенно проще, в Violet поверх этих сервисов предоставляются простые слои абстракции. Например, файл открывается следующим образом:

```
FileService service = FileService.getInstance(initialDirectory);
// detects whether we run under WebStart
FileService.Open open = fileService.open(defaultDirectory, defaultName,
extensionFilter);
InputStream in = open.getInputStream();
String title = open.getName();
```

Интерфейс `FileService.Open` реализуется с помощью двух классов: обертка поверх `JFileChooser` или `FileOpenService` для JNLP.

понравился и игнорировался повсюду. Большинство проектов просто используют собственный сертификат для своих приложений WebStart, что не обеспечивает какой-либо безопасности. Не-приятно, почему разработчики проектов с открытым исходным кодом должны находиться в песочнице JNLP для того, чтобы безопасно опробовать проект.

22.6. Java 2D

В Violet интенсивно используется библиотека Java2D, один из менее известных драгоценных камней в Java API. Каждый узел и ребро имеет метод `getShape`, который выдает `java.awt.Shape` — общий интерфейс всех форм Java2D. В этом интерфейсе реализованы прямоугольники, круги, пути и их объединения, пересечения и разности. Класс `GeneralPath` используется для создания фигур, которые состоят из сегментов произвольных линий и квадратичных/кубических кривых, например, прямых и закругленных стрелок.

Чтобы оценить гибкость Java2D API, рассмотрим следующий код, рисующий тени в методе `AbstractNode.draw`:

```
Shape shape = getShape();
if (shape == null) return;
g2.translate(SHADOW_GAP, SHADOW_GAP);
g2.setColor(SHADOW_COLOR);
g2.fill(shape);
g2.translate(-SHADOW_GAP, -SHADOW_GAP);
g2.setColor(BACKGROUND_COLOR);
g2.fill(shape);
```

Несколько строк кода создают тени для любых форм, даже для тех, которые разработчик может добавить на более позднем этапе.

Конечно, в Violet сохраняются растровые изображения в любом формате, который поддерживается пакетом `javax.imageio`; то есть, GIF, PNG, JPEG, и так далее. Когда мой издатель спросил меня о векторных изображениях, я обнаружил еще одно преимущество библиотеки Java 2D. При печати на PostScript-принтере, Java2D операции преобразуются в операции векторной графики на языке PostScript. Если печатать в файл, результат можно использовать с такими программами, как `ps2eps`, а затем его можно импортировать в Adobe Illustrator или Inkscape. В следующем коде компонент `comp` из Swing, метод которого `paintComponent` используется для рисования графа:

```
DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
String mimeType = "application/postscript";
StreamPrintServiceFactory[] factories;
StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor, mimeType);
FileOutputStream out = new FileOutputStream(fileName);
PrintService service = factories[0].getPrintService(out);
SimpleDoc doc = new SimpleDoc(new Printable() {
    public int print(Graphics g, PageFormat pf, int page) {
```

```

        if (page >= 1) return Printable.NO_SUCH_PAGE;
        else {
            double sf1 = pf.getImageableWidth() / (comp.getWidth() + 1);
            double sf2 = pf.getImageableHeight() / (comp.getHeight() + 1);
            double s = Math.min(sf1, sf2);
            Graphics2D g2 = (Graphics2D) g;
            g2.translate((pf.getWidth() - pf.getImageableWidth()) / 2,
                         (pf.getHeight() - pf.getImageableHeight()) / 2);
            g2.scale(s, s);

            comp.paint(g);
            return Printable.PAGE_EXISTS;
        }
    },
    flavor, null);
DocPrintJob job = service.createPrintJob();
PrintRequestAttributeSet attributes = new HashPrintRequestAttributeSet();
job.print(doc, attributes);

```

В начале, я был обеспокоен тем, что использование общих форм может привести к снижению производительности, но, как оказалось, это не так. Операция вырезания фрагментов работает достаточно хорошо, поскольку в действительности выполняются только те операции, которые необходимы для обновления текущей видимой области.

22.7. Это не фреймворк приложений Swing

В большинстве фреймворков графического пользовательского интерфейса GUI заложена некоторая идея о том, как в приложении происходит управление набором документов с использованием меню, панелей инструментов, строк состояний и т. д. Однако это никогда не было частью Java API. JSR 296 [5] должен был предоставить базовый фреймворк для приложений Swing, но в настоящее время он не действует. Таким образом, у авторов приложений Swing есть два варианта: самостоятельно изобретать велосипед или полагаться на фреймворк сторонних разработчиков. В то время, когда был написан Violet, первыми фреймворками приложений, которые обычно выбирались, были платформы Eclipse и NetBeans, каждая из которых в то время казалась слишком утяжеленной. (В настоящее время есть более широкий выбор, в том числе форки JSR 296, например, GUTS [6]). Таким образом, в Violet потребовалось заново изобретать механизмы для работы с меню и внутренними фреймами.

В Violet вы можете указать пункты меню в файлах свойств, например:

```

file.save.text=Save
file.save.mnemonic=S
file.save.accelerator=ctrl S
file.save.icon=/icons/16x16/save.png

```

Утилита `method` создает пункт меню используя для этого префикс (здесь — `file.save`). Сuffixики `.text`, `.mnemonic` и так далее, являются тем, что сегодня назвали бы «конфигурационным соглашением». Очевидно, что использование файлов ресурсов для описания этих параметров гораздо лучше, чем создание меню с вызовами API, поскольку здесь проще применять локализацию. Я повторно использовал этот механизм в другом проекте с открытым исходным кодом — в среде GridWorld, предназначеннной для изучения информатики в средней школе [7].

Приложения, такие как Violet, позволяют пользователям открывать несколько «документов», в каждой из которых есть граф. Когда был написан первый вариант Violet, то тогда все еще применялся обычный многодокументный интерфейс (MDI). В MDI основной фрейм имел меню, и каждый документа отображается во внутреннем фрейме, имеющим название, но не имеющим меню. Каждый внутренний фрейм находился внутри основного фрейма и пользователь мог изменить его

размер или его минимизировать. Внутренние фреймы могли располагаться каскадно или плитками.

Многим разработчикам не нравится MDI, и поэтому этот стиль пользовательского интерфейса вышел из моды. Какое-то время предпочтение отдавалось однодокументному интерфейсу (SDI), в котором приложение отображает несколько фреймов верхнего уровня; возможно предпочтение отдавалось потому, что этими фреймами можно было управлять с помощью инструментальных средств стандартного окна операционной системы. Когда стало ясно, что наличие большого количества окон верхнего уровня в конце концов не так уж и хорошо, появился интерфейс со вкладками, в котором несколько документов снова находятся в одном фрейме, но теперь все они отображаются в полном размере и выбираются с помощью вкладок. Это не позволяет пользователям сравнивать два документа, помещая их рядом друг с другом, но, кажется, этот подход победил.

Первоначальная версия Violet использовала интерфейс MDI. В Java API есть возможности использовать внутренние фреймы, но я должен был добавить поддержку для размещения фреймов в виде плиток и для каскадного размещения. Александр перешел на интерфейс с вкладками, который несколько лучше поддерживается в Java API. Было бы желательно иметь фреймворк приложения, в котором политика отображения документа была бы прозрачной для разработчика и, возможно, выбиралась бы пользователем.

Александр также добавил поддержку боковых панелей, панели приглашения и заставки. В идеале все это должно быть частью фреймворка приложений на Swing.

22.8. Операции Undo/Redo

Реализация множественных операций undo/redo кажется сложной задачей, но в пакете Swing undo ([Top00], глава 9) приведены хорошие рекомендации, относящиеся к архитектуре. Менеджер UndoManager управляет стеком объектов UndoableEdit. В каждом из них есть метод `undo`, который отменяет эффект операции редактирования, и метод `redo`, который отменяет действие `undo` (то есть, восстанавливает первоначальную операцию редактирования). CompoundEdit представляет собой последовательность операций UndoableEdit, которые должны отменяться или восстанавливаться в полном объеме. Вам предлагается определить небольшие, атомарные операции редактирования (например, добавление или удаление одного ребра или узла в случае использования графа), которые, по мере необходимости, группируются в составные операции редактирования.

Задача состоит в том, чтобы определить небольшой набор атомарных операций, каждую из которых можно легко отменить. В Violet это следующие операции:

- добавление или удаление узла или ребра
- подсоединение или отсоединение потомка узла
- перемещение узла
- изменения свойств узла или ребра

Для каждой из этих операций есть понятное действие отмены `undo`. Например, операция `undo` для добавления узла является удалением узла. Операция `undo` для перемещения узла означает перемещение в обратном направлении.

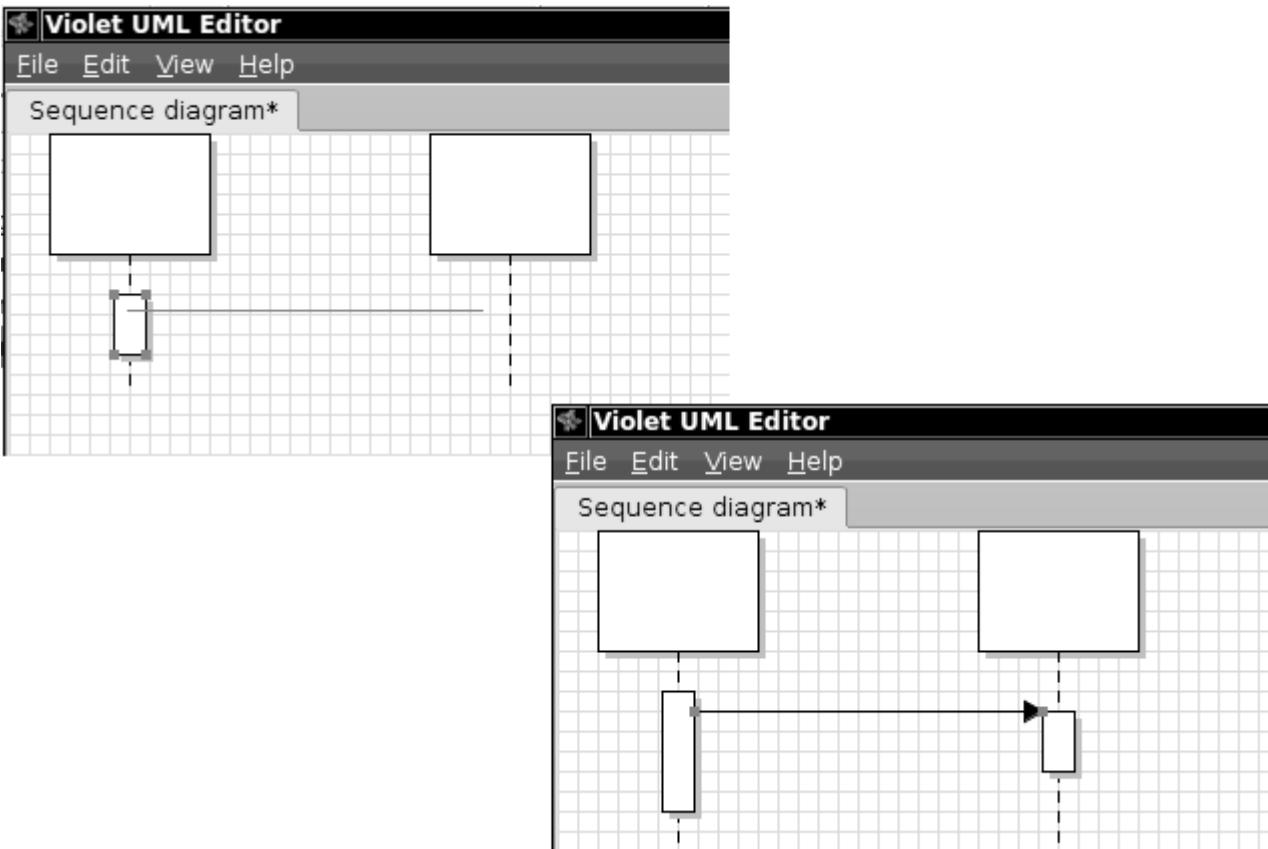


Рис.22.6: Операция Undo должна отменять структурные изменения, сделанные в модели

Отметим, что эти атомарные операции не совпадают с действиями в пользовательском интерфейсе или с методами интерфейса `Graph`, которые вызываются с помощью действий пользовательского интерфейса. Например, рассмотрим диаграмму последовательности на рисунке 22.6, и предположим, что пользователь перетаскивает вправо с помощью мыши блок активации. При отпусканье кнопки мыши, вызывается метод:

```
public boolean addEdgeAtPoints(Edge e, Point2D p1, Point2D p2)
```

Этот метод добавляет ребро, но может также осуществлять другие операции, которые указываются в подклассах `Edge` и `Node`. В этом случае блок активации будет добавлен к линии, расположенной справа. При выполнении операции отмены `undo` нужно также будет удалить блок активации. Таким образом, в *модели* (в нашем случае в графе) должны записываться структурные изменения, которые могут быть отменены. Для этого недостаточно пользоваться только контроллером операций.

Как предусмотрено в пакете `Swing undo`, классы `graph`, `node` и `edge` должны всякий раз, когда происходит структурное изменение, посыпать уведомления `UndoableEditEvent` в менеджер уведомлений `UndoManager`. В *Violet* используется более общая схема, в которой граф сам управляет методами - слушателями (*listeners*) следующего интерфейса:

```
public interface GraphModificationListener
{
    void nodeAdded(Graph g, Node n);
    void nodeRemoved(Graph g, Node n);
    void nodeMoved(Graph g, Node n, double dx, double dy);
    void childAttached(Graph g, int index, Node p, Node c);
    void childDetached(Graph g, int index, Node p, Node c);
    void edgeAdded(Graph g, Edge e);
    void edgeRemoved(Graph g, Edge e);
    void propertyChangedOnNodeOrEdge(Graph g, PropertyChangeEvent event);
}
```

}

Фреймворк устанавливает слушателя в каждом графе; такой слушатель является мостиком к менеджеру операций undo. Чтобы поддерживать операцию undo, нужно дополнительно переопределить общую поддержку слушателей модели — операции, выполняемые с графиком, должны непосредственно взаимодействовать с менеджером undo. Но мне бы также хотелось поддерживать совместимые экспериментальные функции редактирования.

Если вы хотите поддерживать операции undo/redo в вашем приложении, подумайте внимательно об атомарных операциях в модели (а не вашего пользовательского интерфейса). Когда в модели происходят структурные изменения, должны вырабатываться события и менеджер Swing undo должен собирать и группировать такие события.

22.9. Архитектура плагинов

Программисту, знакомому с графикой 2D, будет несложно добавить в Violet новый тип диаграммы. Например, диаграммы активностей были предоставлены третьими лицами. Когда мне потребовалось создать синтаксические диаграммы и диаграммы ER, я решил, что быстрее написать расширения для Violet, а не возиться с Visio или Dia. (На реализацию диаграмм каждого типа потребовалось по одному дню).

Эти реализации не требуются знание всего фреймворка Violet. Необходимы только интерфейсы классов graph, node и edge и соответствующие их реализации. Чтобы упростить задачу разработчикам, я, самостоятельно отделив их от фреймворка, разработал простую архитектуру плагинов.

Конечно, в многих программах есть архитектура плагинов, причем достаточно сложная. Когда кто-то предположил, что Violet должен поддерживать OSGi, я содрогнулся и вместо этого реализовал более простую вещь, которая работает.

Авторы просто создают файл JAR со своими реализациями классов graph, node и edge и помещают его в папку плагинов. Когда Violet запускается, он загружает эти плагины с помощью класса Java ServiceLoader. Этот класс был разработан для загрузки сервисов, например, драйверов JDBC. ServiceLoader загружает файлы JAR, в которых должен быть предоставлен класс, реализующий данный интерфейс (в нашем случае интерфейс Graph).

Каждый файл JAR должен иметь подкаталог META-INF/services, содержащий файл, имя которого является полным квалифицированным именем класса интерфейса (например, com.horstmann.violet.Graph), и в котором в каждой строке указывается имя реализации каждого класса. ServiceLoader строит загрузчик классов для каталога плагинов и загружает все плагины:

```
ServiceLoader<Graph> graphLoader = ServiceLoader.load(Graph.class, classLoader);
for (Graph g : graphLoader) // ServiceLoader<Graph> implements Iterable<Graph>
    registerGraph(g);
```

Это простое, но полезное средство стандартного языка Java, которое может оказаться ценным для ваших собственных проектов.

22.10. Заключение

Как и многие другие проекты с открытым исходным кодом, проект Violet родился вследствие неудовлетворенных потребностей — рисовать простые диаграммы UML с минимальной суетой. Violet стал возможным благодаря удивительной широте платформы Java SE, и в нем используется набор разнообразных технологий, которые являются частью этой платформы. В этой статье я описал, как в Violet используются технологии Java Beans, хранения данных, Java Web Start, Java 2D, Swing

Undo / Redo и средства загрузки сервисов. Не всегда достаточно ясно, как пользоваться этими технологиями в качестве базы Java и Swing, но они могут значительно упростить архитектуру настольных приложений. Они позволили мне, первоначально единственному разработчику, создать в течение нескольких месяцев успешное приложение, работая над ним неполный рабочий день. Использование этих стандартных механизмов позволило предоставить другим разработчикам возможность самостоятельно улучшать Violet, а также применять его отдельные части в своих собственных проектах.

Примечания

1. В то время я еще не знал о замечательной программе UMLGraph, созданной Диомидисом Спинелисом (Diomidis Spinellis) [Spi03]
2. <http://jung.sourceforge.net>
3. <http://www.omg.org/technology/documents/formal/xmi.htm>
4. <http://jcp.org/en/jsr/detail?id=57>
5. <http://jcp.org/en/jsr/detail?id=296>
6. <http://kenai.com/projects/guts>
7. <http://horstmann.com/gridworld>

23. Система VisTrails

Глава 23 из 1 тома книги "[Архитектура приложений с открытым исходным кодом](#)".

Система VisTrails [<http://www.vistrails.org>] является системой с открытым исходным кодом, с помощью которой поддерживаются исследования данных и их визуализация. В ее составе есть постоянно расширяющиеся полезные возможности, предоставляемые в системах научного анализа и визуализации данных. Как и системы научного анализа рабочих процессов, такие как Kepler и Taverna, система VisTrails позволяет в соответствие с набором правил задавать вычислительные процессы, в которых используются существующие приложения, слабо связанные ресурсы и библиотеки. Как и в системах визуализации, таких как AVS и ParaView, в системе VisTrails пользователям предлагаются современные технологии научной и информационной визуализации, позволяющие им исследовать и сравнить различные визуальные представления своих данных. В результате, пользователи могут создавать сложные процессы, которые включают в себя важные этапы научных исследований — от сбора данных и подготовки данных и до манипуляции с комплексным анализом и визуализацией, причем все это интегрировано в одну систему.

Отличительной особенностью системы VisTrails является ее инфраструктура для работы с информацией о происхождении данных. Система VisTrails позволяет по ходу исследовательской задачи собирать данные, получаемые на каждом из шагов, и вести подробную историю их получения. Для автоматизации повторяющихся задач традиционно используются рабочие процессы (workflow), но в приложениях, которые по своему характеру предназначены для исследований, нормой будут очень маленькие и повторяющиеся изменения в этих процессах. По мере того, как пользователь создает и оценивает гипотезы, связанные с его данными, создается ряд различных, но взаимосвязанных рабочих процессов, настройка которых происходит итеративно.

Система VisTrails была разработана для управления этими быстро эволюционирующими рабочими процессами: в ней поддерживается работа с информацией о происхождении продуцируемых данных (например, визуализация, графики), о рабочих процессах, в которых эти данные продуцируются, и о выполнении этих рабочих процессов. В системе также есть средства аннотации, так что пользователи могут автоматически собираемые данные дополнять описаниями.

Кроме возможности повторно воспроизводить результаты, система VisTrails упрощает доступ к информации о происхождении данных с помощью ряда операций и интуитивно понятного интерфейса, в котором пользователям предоставляется возможность совместно анализировать данные.

Примечательно, что благодаря тому, что в системе есть возможность хранить временные результаты, в ней поддерживается работа с рефлексивными рассуждениями, позволяющими пользователям изучить действия, ведущие к результату, и отслеживать цепочки рассуждений как в прямом, так и в обратном направлении. Пользователи могут интуитивно понятным способом переходить от одной версии рабочего процесса к другой, отменять изменения, не теряя при этом результатов, визуально сравнивать несколько рабочих процессов и одновременно показывать их результаты в таблице визуализации.

В VisTrails затронуты важные вопросы удобства использования системы (usability), препятствующие более широкому распространению систем на базе рабочих процессов и систем визуализации. Для удобства широкого круга пользователей, в том числе тех, у кого нет опыта в области программирования, в системе предоставляется возможность использовать серии действий и интерфейсы, упрощающие создание и использование рабочих процессов, в том числе позволяющие в интерактивном режиме по рекомендациям, предлагаемым системой, создавать и улучшать рабочие процессы по аналогии, запрашивать рабочие процессы по примерам, а также автоматически завершать разработку рабочих процессов. Мы также разработали новый фреймворк, позволяющий создавать специализированные приложения, которые могут легко устанавливаться и настраиваться конечными пользователями (не экспертами).

Возможность расширения системы VisTrails обусловлена ее инфраструктурой, которая позволяет пользователям легко интегрировать в систему другие инструментальные средства и библиотеки, а также быстро создать прототипы новых функций. Эта возможность сыграла важную роль в создании условий, позволяющих использовать систему в различных прикладных областях, в том числе в экологических исследованиях, психиатрии, астрономии, космологии, физики высоких энергий, квантовой физике и молекулярном моделировании.

Чтобы система VisTrails оставалась для всех бесплатной системой с открытым исходным кодом, мы создавали ее только с использованием бесплатных пакетов, имеющих открытый исходный код. Система VisTrails написана на языке Python, а в качестве инструментального средства графического интерфейса используется Qt (через привязку PyQt Python). Чтобы увеличить количество пользователей и расширить спектр используемых приложений, мы, помня о требовании переносимости, разрабатывали систему с нуля. Система VisTrails работает на платформах Windows, Mac и Linux.

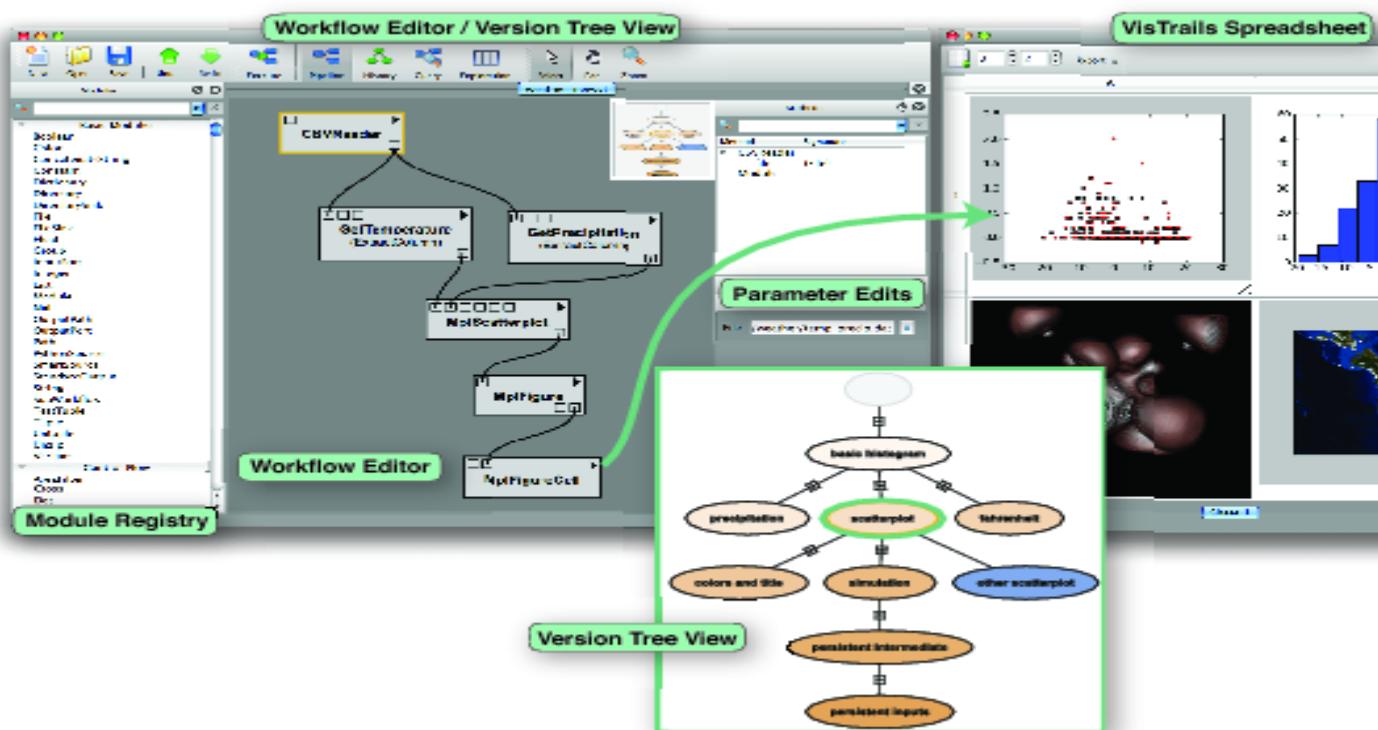


Рис.23.1: Компоненты пользовательского интерфейса системы VisTrails

23.1. Обзор системы

Исследование данных является по своей сути творческим процессом, который требует от пользователей искать относящиеся к делу данные, собрать и визуализировать эти данные, общаться с коллегами, исследовать различные решения, и рассылать получаемые результаты. Если учесть размер данных и сложность анализа, которые обычно применяются в научных исследованиях, нужны инструментальные средства, которые будут лучшей поддержкой творчеству.

К таким инструментальным средствам предъявляются два неразрывно связанных друг с другом основных требования. Во-первых, важно иметь возможность с помощью формального описания определять исследовательские процессы, которые, в идеале, являются исполняемыми процессами. Во-вторых, для того, чтобы воспроизводить результаты этих процессов, а также рассуждения, делающие на различных этапах последовательного решения проблемы, эти инструментальные средства должны иметь возможность систематическую собирать информацию о происхождении данных. Система VisTrails была разработана с учетом этих требований.

23.1.1. Рабочие процессы и системы на базе рабочих процессов

В системах с рабочими процессами (workflow systems) поддерживается создание конвейеров или рабочих процессов (workflows), с помощью которых происходит объединение нескольких инструментальных средств. Как таковые, они позволяют автоматизировать повторяющиеся задачи и повторно воспроизводить полученные результаты. Рабочие потоки являются скриптами командных оболочек, в которых можно быстро занять примитивы. Они предназначены для решения широкого спектра задач, о чем свидетельствует ряд приложений, в которых применяются рабочие процессы, причем как коммерческие (например, Apple Mac OS X Automator и Yahoo! Pipes), так и академические (например, NiPype, Kepler и Taverna).

Рабочие процессы имеют ряд преимуществ в сравнение со скриптами и программами, написанными на языках высокого уровня. Они предлагают простую модель программирования, в которой последовательность задач создается с помощью подключения выходов одной задачи к входу другой. На рис 23.1 показан рабочий процесс который читает файл CSV, содержащий метеорологические наблюдения, и создает диаграммы разброса значений.

Такая упрощенная модель программирования позволяет системам на базе рабочих процессов использовать интуитивно понятный визуальный интерфейс программирования, что делает их более удобными для пользователей, не обладающих существенным опытом программирования. Рабочие процессы также имеют понятную структуру: их можно изображать в виде графов, в которых узлы представляют собой процессы (или модули) вместе с их параметрами, а с помощью ребер представлен поток данных между процессами. В примере на рис.23.1 модуль CSVReader получает в качестве параметра имя файла (`/weather/temp_precip.dat`), читает файл, и передает его содержимое в модули GetTemperature и GetPrecipitation, которые, в свою очередь, отправляют значения температуры и осадков в функцию matplotlib, с помощью которой создается график разброса значений.

Большинство систем с рабочими потоками предназначены для конкретных областей применения. Например, назначение системы Taverna — рабочие процессы из области биоинформатики, а система NiPype позволяет создавать рабочие потоки для нейровизуализации. Хотя в системе VisTrails поддерживается большинство функциональных возможностей, предоставляемых другими системами с рабочими потоками, в ней, система VisTrails благодаря тому, что в ней интегрировано большое количество инструментальных средств, библиотек и сервисов, разработана для поддержки исследовательских задач общего назначения из широкого спектра областей применения.

23.1.3. Информация о происхождении данных и рабочих процессов

В научном сообществе хорошо известно, что важно сохранять информацию о происхождении результатов (и вычислении данных). Эта информация (также называемая аудиторской информацией, информацией о происхождении или родословной), в которой указываются сведения о процессе и данных, используемых при получении результатов исследования. Информация о происхождении данных является ключевой при архивации данных, при оценке качества данных и определении авторства, при воспроизведении результатов, а также при их проверке.

Важным компонентом этих данных является информация о причинно следственных зависимостях (*causality — казуальности*), т. е. описание процесса (последовательности шагов), который вместе с входными данными и параметрами привел к получению конкретных результатов. Таким образом, в структуре информации об источнике данных отражается структура рабочего процесса (или набор процессов), используемого для получения данного результирующего набора данных.

На самом деле, в научных исследованиях катализатором широкого использования систем на базе рабочих процессов явилось то, что их легко было использовать для автоматического сохранения информации о происхождении данных. Ранее существовавшие системы были *расширены* с тем, чтобы можно было собирать информацию о происхождении данных, а система VisTrails была *изначально разработана* для поддержки работы с информацией о происхождении данных.

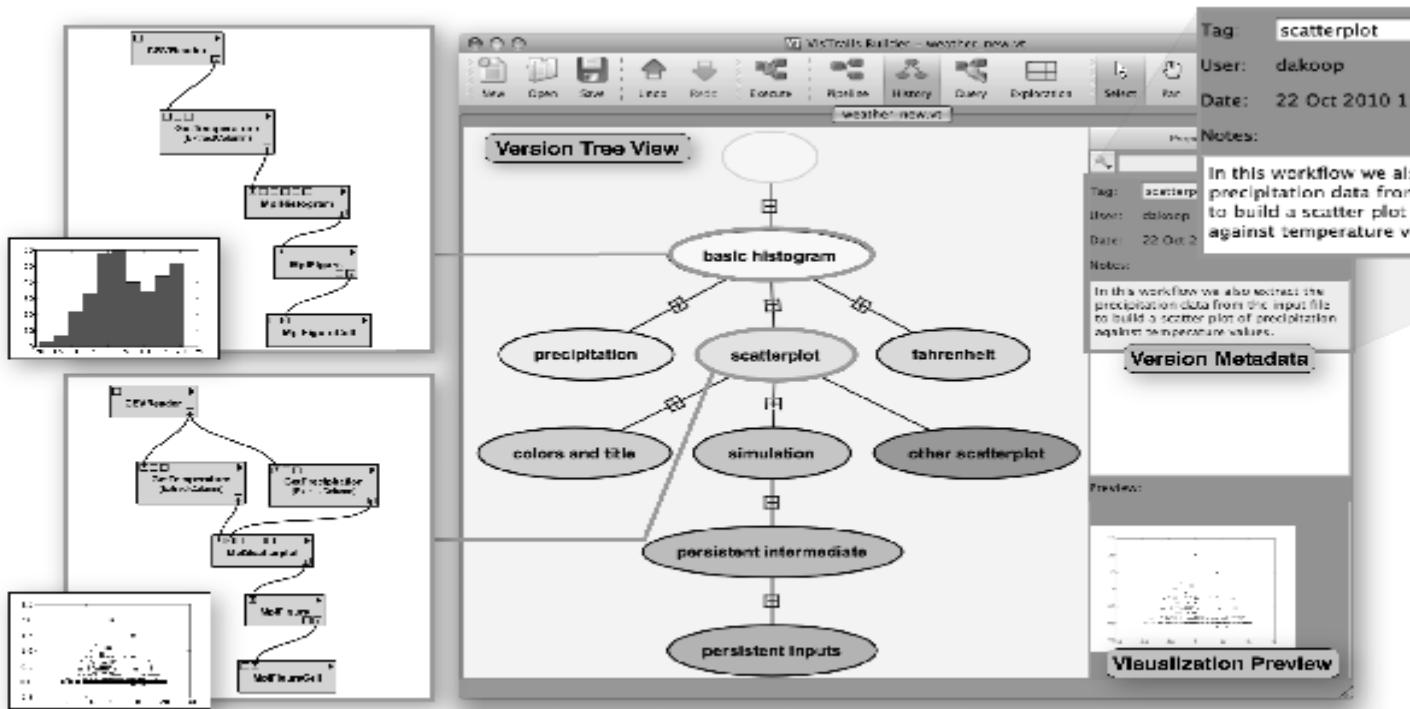


Рис.23.2: Информация о происхождении данных, снабженная аннотациями

23.1.3. Пользовательский интерфейс и базовые функциональные возможности

На рис.23.1 и рис.23.2 показаны различные компоненты пользовательского интерфейса системы. Пользователи с помощью редактора рабочих процессов Workflow Editor создают и редактируют рабочие процессы.

Чтобы создавать графы рабочих процессов, пользователи могут перетаскивать модули из реестра модулей Module Registry в канвас редактора рабочих процессов Workflow Editor. В системе VisTrails предложены наборы встроенных модулей, а пользователям также разрешается добавлять свои собственные модули (подробности сммотрите в разделе 23.3). Когда модуль выбран, система

VisTrails отображает его параметры (в области редактирования параметров Parameter Edits), и пользователь может задать их значения или их изменить.

По мере того, как спецификация рабочего процесса уточняется, система сохраняет изменения и представляет их пользователю для просмотра в окне дерева версий Version Tree View, которое будет описано ниже. Пользователи могут взаимодействовать с рабочими процессами и результатами их работы через таблицу VisTrails Spreadsheet. Каждая ячейка таблицы представляет собой окно, которое соответствует некоторому экземпляру рабочего процесса. На рис.23.1 результаты работы рабочего процесса показаны в редакторе рабочих процессов Workflow Editor, который показан в верхней левой ячейке таблицы. Пользователи могут либо напрямую изменять параметры рабочего процесса, либо синхронизировать их с содержимым различных ячеек электронной таблицы.

Окно дерева версий Version Tree View помогает пользователям переходить от одних версий рабочего процесса к другим. Как видно на рис.23.2, пользователь, щелкнув мышкой по узлу в дереве версий, может увидеть рабочий процесс, связанный с ним результат (режим предварительного просмотра Visualization Preview) и метаданные. Некоторые из метаданных сохраняются автоматически, например, идентификатор пользователя, который создал конкретный рабочий процесс, и дата его создания, но пользователи также могут предоставлять дополнительные метаданные, в том числе заполнить тег, идентифицирующий рабочий процесс, и ввести его текстовое описание.



- Создание/редактирование рабочего процесса
- Выполнение рабочего процесса
- Сохранение рабочих процессов и информации о происхождении данных

- Запросы версий, рабочих процессов, информации о происхождении данных
- Публикация в статьях и на веб-страницах

Рис.23.3: Архитектура системы VisTrails

23.2. История проекта

Первые версии системы VisTrails были написаны на языках Java и C++. Версия на языке C++ была предложена некоторым самим первым пользователям, обратная связь с которыми сыграла важную роль в формировании наших требований к системе.

Наблюдая в ряде научных сообществ тенденцию увеличения количества библиотек и инструментальных средств, в которых используется язык Python, мы решили использовать этот язык в качестве основы системы VisTrails. Язык Python быстро становится универсальным современным средством, позволяющим состыковывать программное обеспечение, предназначенное для научных разработок. Во многих библиотеках, написанных на разных языках, таких как Fortran, С и С++, применяются привязки языка Python в качестве средства написания скриптов. Т.к. система VisTrails предназначается для объединения в рабочих процессах в единый оркестр большого количества различных библиотек, то делать это будет проще в случае, если система реализована на чистом языке Python. В частности, в языке Python есть возможности динамической загрузки кода, напоминающие те, которые присутствуют в среде языка LISP, у которой существенно большее сообщество разработчиков и исключительно богатая стандартная библиотека. В конце 2005 года мы приступили к разработке текущей системы, использующей Python/PyQt/Qt. Такой выбор значительно упростил создание в системе расширений, в частности, добавление новых модулей и пакетов.

Бета-версия системы VisTrails была впервые выпущена в январе 2007 года. С тех пор система была скачана более двадцати пяти тысяч раз.

23.3. Внутри системы VisTrails

На рис.23.3 показана схема архитектуры системы VisTrails, изображающая внутренние компоненты, в которых осуществляется поддержка функций пользовательского интерфейса, описанных выше. Выполнение рабочего процесса осуществляется под управлением движка Execution Engine, с помощью которого происходит отслеживание вызываемых операций и используемых с ними параметров и выполняется сохранение информации о ходе выполнения рабочего процесса (информация о происхождении данных в ходе выполнения процесса). Система VisTrails также позволяет во время выполнения процесса кэшировать промежуточные результаты в памяти и/или на диске. Как мы покажем в разделе 23.3, повторный запуск происходит только для новых комбинаций модулей и параметров, причем выполнение происходит с помощью обращений к функциям, лежащим глубже (например, к matplotlib). После этого результаты выполнения рабочего процесса вместе с исходными данными о происхождении данных могут добавляться в электронные документы (раздел 23.4).

Информация об изменениях в рабочих процессах накапливается в дереве версий Version Tree, которое может храниться в различных хранилищах данных, в том числе в хранилище файлов XML в локальном каталоге или в реляционной базе данных. В системе VisTrails также предоставляется механизм обработки запросов, который позволяет пользователям изучать информацию о происхождении данных.

Отметим, что, хотя система VisTrails была разработана как интерактивный инструмент, ее также можно использовать в режиме сервера. Как только рабочие процессы будут созданы, их можно будет выполнить на сервере VisTrails. Эта возможность полезна в ряде сценариев, в том числе при создании веб-интерфейса, в котором пользователям дается возможность взаимодействовать с рабочими процессами и запускать рабочие процессы в высокопроизводительных вычислительных средах.

23.3.1. Дерево версий: возможность выбора источника данных

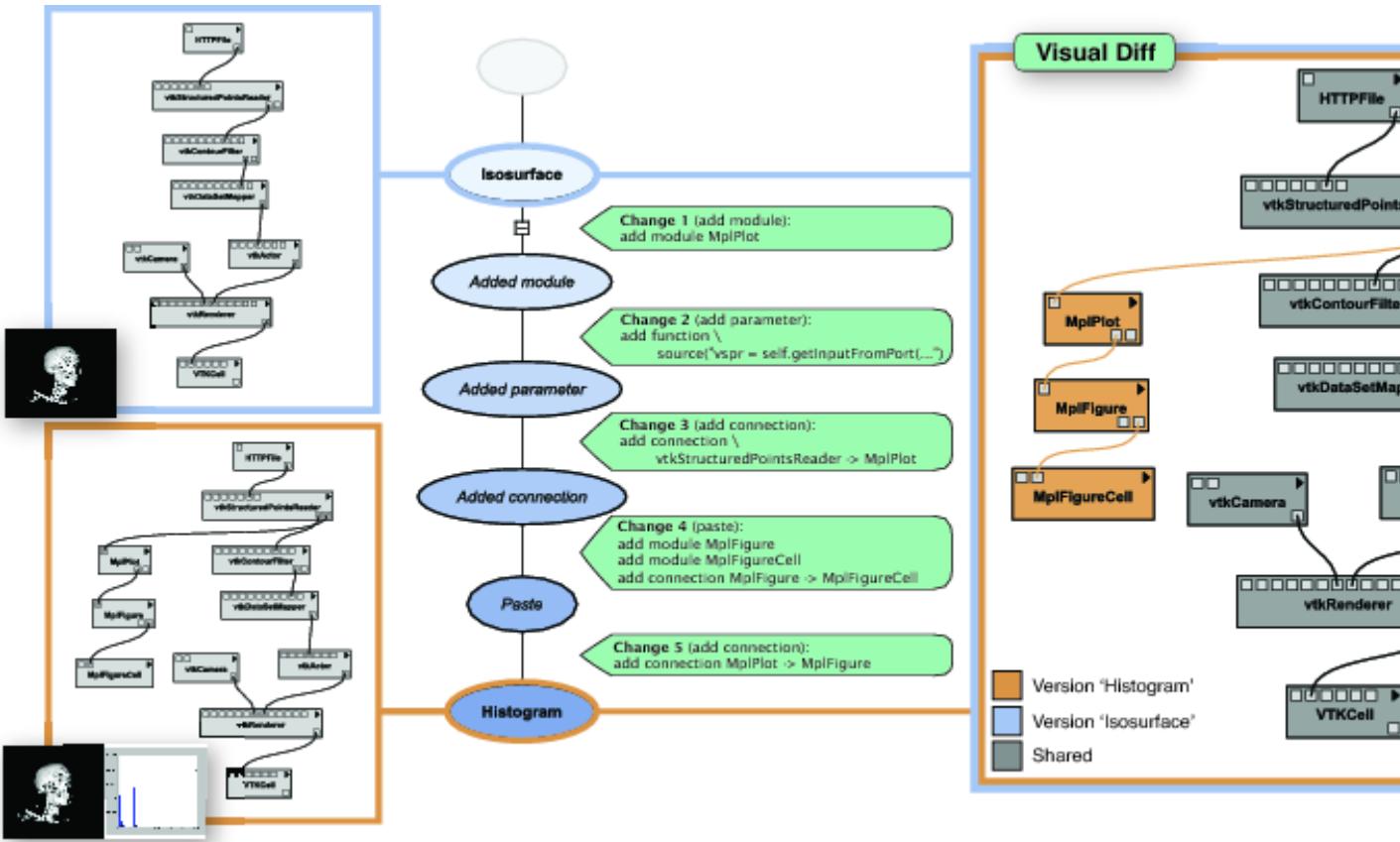


Рис.23.4: Модель происхождения, основанная на изменениях

Новой концепцией, которая была введена с системой VisTrails, является понятие происхождения эволюции рабочего процесса. В отличие от предыдущих систем визуализации, использующих или созданных основе рабочих процессов, в которых понятие происхождения поддерживается только для полученных результатов обработки данных, в системе VisTrails рабочие процессы трактуются как элементы данных первого порядка и сохраняется информация об их происхождении. Наличие понятия эволюции происхождения рабочего процесса поддерживает использование рефлексивных рассуждений. Пользователи могут сразу изучать большое количество цепочек рассуждений, не теряя при этом результатов, а поскольку система сохраняет промежуточные результаты, пользователи могут использовать эту информацию при рассуждениях и делать из нее выводы. В результате также появляется возможность выполнять серии операций, благодаря которым процессы исследований упрощаются. Например, пользователи могут легко перемещаться по пространству рабочих процессов, созданных для данной задачи, визуально сравнивать рабочие процессы и их результаты (смотрите рис 23.4), а также исследовать (огромные) пространства параметров. Кроме того, пользователи могут запрашивать информацию о происхождении и обучаться на собственных примерах.

Информация об эволюции рабочих потоков собирается при помощи модели происхождения, базирующейся на изменениях. Как показано на рисунке 23.4, в системе VisTrails операции или изменения, которые применяются к рабочим процессам (например, добавление модуля, изменение параметров и т. д.), хранятся точно также, как транзакции в журнале транзакций баз данных. Эта информация сохраняется в виде дерева, в котором каждый узел соответствует версии рабочего потока, а ребро между родительским и дочерним узлом, представляет собой изменение, которое было применено к родительскому узлу для того, чтобы получить дочерний узел. При рассмотрении этого дерева мы пользуемся терминами «дерево версий» и *vistrail* (сокращение от *visual trail* — визуальный след) как взаимозаменяемыми. Обратите внимание, что модель, базирующаяся на изменениях, одинаковым образом сохраняет изменения значений параметров и изменения определений

рабочего потока. Такой последовательности изменений достаточно, чтобы определить происхождение получаемых результатов данных, а также собрать информацию том, как с течением времени развивается рабочий процесс. Модель проста и компактна — для нее требуется существенно меньше места, чем альтернативному варианту хранения нескольких *версий* рабочих потоков.

Есть ряд преимуществ, которые можно получить при использовании такой модели. На рис.23.4 показаны функциональные возможности визуального отображения различий, которые в системе VisTrails предоставляются при сравнении двух рабочих процессов. Хотя процессы представлены в виде графов, использующих модель, основанную на изменениях, сравнение двух рабочих процессов становится очень простым: достаточно пройти по дереву версий и идентифицировать последовательность действий, которая потребовалась бы для того, чтобы преобразовать один рабочий процесс в другой.

Другим важным преимуществом модели происхождения, основанной на изменениях, является то, что используемое дерево версий может использоваться в качестве механизма поддержки совместной работы. Поскольку разработка рабочих процессов является крайне сложной задачей, она часто требует участие в работе сразу нескольких пользователей. С помощью дерева версий не только предоставляется интуитивно понятный способ визуализации вклада различных пользователей (например, с помощью окрашивания узлов в зависимости от того, какой пользователь создал соответствующий рабочий процесс), но и благодаря однородности модели можно использовать простые алгоритмы, позволяющие синхронизировать изменения, выполняемые многими пользователями.

Во время выполнения рабочего процесса можно легко собрать и сохранить информацию о происхождении данных. После того, как выполнение будет завершено, также важно сохранить взаимосвязь между полученными данными и информацией о их происхождении, т.е. о том, какой рабочий процесс, какие параметры и какие исходные файлы использовались для получения результирующих данных. Если файлы данных или информация о происхождении данных перемещаются или изменяются, то становится трудно найти данные, связанные с информацией о происхождении, или найти информацию о происхождении, связанную с данными. В системе VisTrails предоставляется механизм долговременного хранения данных, с помощью которого осуществляется управление входными, промежуточными и выходными файлами данных, а также сохраняется взаимосвязь между информацией о происхождении данных и данными. Этот механизм обеспечивает лучшую поддержку при выполнении повторных вычислений, поскольку с его помощью гарантируется, что можно найти данные (и они будут правильными), указываемые в информации о происхождении. Другим важным преимуществом такого управления является то, что есть возможность кэшировать промежуточные данные, к которым затем также могут обращаться другие пользователи.

23.3.2. Выполнение рабочего процесса, кэширование данных

Двигок в системе VisTrails, предназначенный для выполнения рабочего процесса, был разработан таким образом, чтобы можно было интегрировать в систему новые и уже существующие инструментальные средства и библиотеки. Мы постарались учсть различные стили, обычно используемые при добавлении научного программного обеспечения, созданного сторонними разработчиками и предназначенного для визуализации и вычислений. В частности, система VisTrails может быть интегрирована с прикладными библиотеками, которые существуют либо в виде предварительно скомпилированных двоичных файлов, выполняются в оболочке и в качестве входа и выхода используют файлы, либо в виде библиотек классов языков C++ / Java / Python, которым в качестве входа и выхода передаются внутренние объекты.

В системе VisTrails адаптирована модель исполнения потоков данных, в которой каждый модуль выполняет вычисления и данные, создаваемые модулем, перемещаются по соединениям, которые существуют между модулями. Модули выполняются по принципу снизу-вверх; каждый набор входных данных создается по требованию путем рекурсивного выполнения модулей, находящихся выше (мы говорим, что модуль А находится выше модуля В, если есть последовательность соединений, ведущая от А к В). Промежуточные данные временно хранятся либо в памяти (как объект

языка Python) или на диске (объект-оболочка на языке Python, в котором содержится информация о доступе к данным).

Чтобы разрешить пользователям добавлять в систему VisTrails свои собственные функции, мы собрали расширяемую систему пакетов (смотрите раздел 23.3). Система пакетов позволяет пользователям включать в рабочий процесс их собственные или сторонние модули. Разработчик пакета должен определить набор вычислительных модулей и для каждого модуля задать входные и выходные порты, а также определить само вычисление. Для существующих библиотек нужно, чтобы в методе вычисления был указан перевод из входных портов в параметры существующих функций и отображение результирующих значений в выходные порты.

В поисковых задачах, подобные рабочие процессы, которые совместно используют общие подструктуры, часто выполняются в строгой последовательности. Для того, чтобы повысить эффективность выполнения рабочего процесса, промежуточные результаты в системе VisTrails кэшируются с целью свести к минимуму повторные вычисления,. Поскольку мы повторно пользуемся предыдущими результатами вычислений, мы неявно предполагаем, что будут использоваться модули кэширования: для них то, что было на входе, будет и на выходе. Это требование накладывает определенные ограничения на поведение классов, но мы считаем, что они обоснованы.

Но есть очевидные ситуации, в которых такого поведения достигнуть не удается. Например, модуль, который загружает файл на удаленный сервер или сохраняет файл на диске, обладает существенным побочным эффектом, тогда то, что он выдает в качестве выходных данных, имеет сравнительно небольшое значение. В других модулях может использоваться рандомизация, и их недeterminизм может быть желательным; такие модули можно пометить как некэшируемые. Однако, некоторые модули, которые, в сущности, не являются функциональными, могут быть преобразованы; функция, которая записывает данные в два файла, может быть представлена как выдающая содержимое этих файлов.

23.3. Внутри системы VisTrails

23.3.3. СерIALIZАЦИЯ И ХРАНЕНИЕ ДАННЫХ

Одним из ключевых компонентов любой системы, поддерживающей использование информации о происхождении, является сериализация и хранение данных. В системе VisTrails данные изначально сохраняются в формате XML с использованием простых методов `fromXML` и `toXML`, встроенных во внутренние объекты системы (например, в дерево версий, в каждый модуль). Чтобы поддерживать эволюцию схемы подобных объектов, в этих функциях также закодированы все переходы между версиями схемы. По мере того, как проект разрабатывался, база наших пользователей росла, и мы решили поддерживать различные версии сериализации, в том числе и реляционные хранилища. В добавок по мере того, как добавлялись объекты схемы, нам для управления обычными данными потребовалось поддерживать более развитую инфраструктуру, позволяющую управлять версиями схем, выполнять преобразование между версиями и работать с существенными отношениями. Чтобы все это сделать, мы добавили новый слой базы данных (`db`).

Слой (`db`) состоит из трех основных компонентов: доменных объектов, сервис-логики и методов сохранения данных. Доменная компонента и компонента сохранения данных позволяют использовать версии, так что для каждой версии схемы есть свой собственный набор классов. Таким образом, у нас поддерживается код, позволяющий читать каждую версию схемы. Также есть классы, в которых определены правила перевода объектов из одной версии схемы в другие. В классах сервис-логики предоставляются методы взаимосвязи с данными и методы, осуществляющие обнаружение и преобразование версий схемы.

Поскольку написание большей части этого кода утомительно и повторяющееся, мы используем шаблоны и мета-схемы, с помощью которых определяется как компоновка объекта (и все индексы

доступа к памяти), так и код сериализации. Мета-схема написана на языке XML и является расширяемой в том смысле, что можно добавлять варианты сериализаций, отличные используемых по умолчанию XML и реляционных отображений, определенных в системе VisTrails. Этот подход похож на объектно-реляционные отображения и фреймверки, например, Hibernate [2] и SQLObject [3], но лишь добавлено несколько специальных процедур, которые автоматизируют такие задачи, как повторное отображение идентификаторов и преобразование объектов из одной версии схемы в следующую. Кроме того, мы также можем использовать те же самые мета-схемы при генерации кода сериализации для многих языков. Сначала была написана мета-схема meta-Python, в которой код домена и код сохранения данных были сгенерированы с помощью выполнения кода на языке Python с переменными, полученными из мета-схемы, а совсем недавно мы перешли к шаблонам Mako [4].

Автоматическое преобразование является ключевым для пользователей, которым нужно перенести свои данные на новые версии системы. В нашем проекте были добавлены специальные возможности, которые сделали для разработчиков такое преобразование чуть менее болезненным. Поскольку мы поддерживаем копии кода для каждой версии, перевод кода нужен просто для отображения одной версии в другую. На корневом уровне, мы определяем отображение, в котором указывается, каким образом любая версия может быть преобразована в любую другую версию. Для версий, которые сильно разнесены друг от друга, обычно указывается цепочка преобразований через несколько промежуточных версий. Сначала использовалось отображение только в прямом направлении, а это означает, что новые версии не могут преобразовываться в старые, но в совсем недавних отображениях схем было добавлено отображение в обратном направлении.

В каждом объекте есть метод `update_version`, который получает произвольную версию объекта и возвращает текущую версию. По умолчанию, он выполняет рекурсивное преобразование, при котором каждый объект обновляется при помощи отображения полей старого объекта в те, которые есть в новой версии. По умолчанию это отображение представляет собой копирование каждого поля с тем же самым именем, но с помощью метода, указываемого как "override" ("переопределяющий"), можно для любого поля можно переопределить поведение, определенное по умолчанию. Переопределяющий метод является методом, который берет старый объект и возвращает новую версию. Поскольку большинство изменений в схеме затрагивает только небольшое количество полей, в большинстве случаев достаточно отображения, выполняемого по умолчанию, но переопределения предоставляют гибкий механизм внесения локальных изменений.

23.3.4. Расширяемость с помощью пакетов и языка Python

В первом прототипе системы VisTrails был фиксированный набор модулей. Это была идеальная среда для разработки базовых идей, связанных с деревом версий VisTrails и кэширование многочисленных запусков рабочих процессов, но с точки зрения долгосрочной перспективы прототип был сильно ограничен.

Мы рассматриваем систему VisTrails как инфраструктуру для вычислений, а это, в буквальном смысле, значит, что в системе должны быть вспомогательные подмостки для других инструментальных средств и процессов, которые должны быть разработаны. Главное требование этого сценария является расширяемость. Типичный способ для достижения этой цели включает в себя определение целевого языка и написание соответствующего интерпретатора. Это выглядит привлекательным благодаря очень индивидуальному контролю, который предоставляется за исполнением. Эта привлекательность усиливается в свете наших требований кэширования. Тем не менее, внедрение полноценного языка программирования требует больших усилий, что никогда не было нашей главной целью. Что еще более важно, это заставит пользователей, которые просто пытаются использовать систему VisTrails, изучать совершенно новый язык, который им не нужен.

Нам нужна была система, которая позволяла пользователям легко добавлять свои собственные функции. В то же время, нам нужно, чтобы система была достаточно мощной, с тем чтобы самостоятельно реализовывать довольно сложные части программного обеспечения. В качестве при-

мера, в системе VisTrails поддерживается библиотека визуализации VTK [5]. В VTK содержится около 1000 классов, которые изменяются в зависимости от вариантов компиляции, конфигурирования и операционной системы. Поскольку нам показалось, что будет контрпродуктивным и в конечном счете безнадежной писать разные пути кода для всех этих случаев, мы решили, что набор модулей VisTrails, который предоставляется в любом заданном пакете, нужно определять динамически и, естественно, библиотека VTK, стала нашей целевой моделью для сложных пакетов.

Вычислительные науки были одной из областей, для использования в которых мы изначально назначивали систему, и когда мы разрабатывали ее, среди этих ученых языка Python становился в качестве "кода для склейки". Поскольку при спецификации поведения модулей VisTrails, определяемых пользователями, используется сам язык Python, мы могли бы справиться со всем, но преодолели один большой барьер. Как оказалось, в языке Python предлагается замечательная инфраструктура для динамически определяемых классов и рефлексии. Почти каждое определение в языке Python имеет эквивалентную форму в виде выражения первого порядка. Для системы наших пакетов особенно важными возможностями рефлексии языка Python являются следующие две:

- Классы языка Python могут определяться с помощью вызова динамического квалификатора `type`. Возвращаемым значением является представление класса, который можно использовать точно так же, как и класс языка Python, определяемый обычным образом.
- Модули языка Python можно импортировать с помощью вызовов функции `__import__`, и результирующее значение ведет себя точно так же, как идентификатор в стандартной инструкции `import`. Путь, используемый для получения этих модулей, можно указывать время выполнения.

Конечно, использование языка Python в качестве целевого, имеет ряд недостатков. Прежде всего, такая динамическая природа языка Python означает, что пока мы не захотим реализовать некоторые вещи, такие как безопасность типов пакетов VisTrails, они вообще будут невозможны. Более того, некоторые из требований к модулям VisTrails, в частности такие, как прозрачность при ссылках на модули (подробнее об этом позже), не могут поддерживаться в Python. Тем не менее, мы считаем, что с помощью искусственных механизмов целесообразно ограничить конструкции, допустимые в языке Python, и с этой оговоркой, язык Python является чрезвычайно привлекательным языком, позволяющим расширять программное обеспечения.

23.3.5. Пакеты и сборки системы VisTrails

Пакет системы VisTrails инкапсулирует в себе набор модулей. Его самое обычное представление на диске точно такое, как представление пакета Python (к несчастью, допускающего коллизию имен). Пакет Python состоит из набора файлов Python, в которых определяются функции и классы языка Python. Пакет системы VisTrails является пакетом языка Python, в котором предпочтение отдается определенному интерфейсу. В нем есть файлы, в которых определены конкретные функции и переменные. В своей простейшей форме, пакет VisTrails должен быть каталогом, содержащим два файла: `__init__.py` и `init.py`.

Первый файл `__init__.py` добавлен в соответствие с требованиям к пакетам Python, в нем должны быть только несколько определений, являющиеся константами. Хотя нет никакого способа это проконтролировать, пакеты VisTrails, в которых не соблюдаются это требование, рассматриваются как ошибочные. Среди значений, определенных в файле, есть уникальный глобальный идентификатор пакета, который используется для того, чтобы различать модули, когда происходит их сериализация или создаются их версии (версии пакета важны при обработке рабочего процесса и обновлении пакетов; смотрите раздел 23.4). В этом файле также могут быть функции `package_dependencies` и `package_requirements`. Поскольку в модулях VisTrails допускаются подклассы других модулей VisTrails, не входящих в корневой класс `Module`, вполне возможно, что в некотором пакете VisTrails есть расширение поведения другого пакета, и, поэтому, первый пакет должен быть проинициализирован перед вторым пакетом. Такие взаимозависимости пакетов определяются с помощью `package_dependencies`. С другой стороны, с помощью функции

`package_requirements` определяются требования к библиотекам системного уровня, которые в системе VisTrails могут, в некоторых случаях, выполняться автоматически за счет использования абстракции сборок.

Сборка (bundle) является пакетом системного уровня, управление которым в системе VisTrails происходит при помощи специальных системных инструментальных средств, таких как RPM в RedHat или APT в Ubuntu. Когда все эти особенности соблюдены, система VisTrails может определить свойства пакета с помощью непосредственного импорта модуля Python и доступа к соответствующим переменным.

Во втором файле, `init.py`, находятся точки входа для всех актуальных определений модулей VisTrails. Наиболее важной особенностью этого файла является определение двух функций `initialize` и `finalize`. Функция `initialize` вызывается тогда, когда пакет становится доступным после того, как стали доступными все пакеты, от которых он зависит. С ее помощью выполняются задачи настройки для всех модулей в пакете. С другой стороны, функция `finalize` обычно используется, чтобы освободить ресурсы времени выполнения (например, могут быть стерты временные файлы, созданные в пакете).

Каждый модуль системы VisTrails представлен в пакете в виде одного класса Python. Чтобы зарегистрировать этот класс в системе VisTrails, разработчик пакетов один раз для каждого модуля VisTrails обращается к функции `add_module`. Такие модули VisTrails могут быть любыми классами языка Python, но в них должны соблюдаться некоторые требования. Первое из них то, что каждый из этих классов должен быть подклассом базового класса Python, определенным в системе VisTrails, хотя это может показаться скучным, обращением к `Module`. В модулях VisTrails может использоваться множественное наследование, но только один из классов должен быть модулем VisTrails — решетчатые иерархии в дереве модулей VisTrails не допускаются. Множественное наследование становится полезным, в частности, при определении смешанных классов: простое поведение, закодированное в родительских классах, может быть объединено вместе с тем, чтобы создать более сложное поведение.

Набор имеющихся портов определяет интерфейс модуля VisTrails, так что это влияет не только на отображение этих модули, но и взаимодействие этих модулей с другими. Итак, эти порты должны быть явно описаны в инфраструктуре системы VisTrails. Это может быть сделано либо путем соответствующих обращений к функциям `add_input_port` и `add_output_port` при обращении к функции `initialize`, либо указав списки `add_input_port` и `add_output_port` в каждом модуле VisTrails для каждого класса.

В каждом модуле вычисления, которые должны быть выполнены, определяются при помощи переопределения метода `compute`. Данные передаются между модулями через порты и доступ к ним осуществляется через методы `get_input_from_port` и `set_result`. В традиционной среде потоков данных, порядок выполнения определяется по требованию при помощи запросов данных. В нашем случае, порядок выполнения определяется при помощи топологической сортировки модулей рабочего процесса. Поскольку для алгоритма кэширования требуется ациклический граф, мы составляем план выполнения в обратном топологическом порядке сортировки, с тем чтобы обращения к этим функциям не требовали переключения на исполнение модулей, находящихся выше. Мы приняли это решение сознательно: оно позволяет проще рассматривать поведение каждого модуля вне зависимости от всех остальных модулей, что делает нашу стратегию кэширования более простой и надежной.

В качестве общей рекомендации, в модулях системы VisTrails нужно при переопределении модуля `compute` воздерживаться от использования функций с побочными эффектами. Как отмечалось в разделе 23.3, благодаря этому требованию становится возможным кэширование конкретного рабочего процесса: если модуль отвечает этому требованию, то его поведение является функцией, зависящей от выходных данных модулей, находящихся выше. Тогда для каждого ациклического

подграфа вычисления можно делать только один раз, и результатами можно пользоваться повторно.

23.3.6. Передача данных в виде модулей

Одной особенностью модулей VisTrails и их взаимодействия является то, что данные, которые передаются между модулями VisTrails, сами являются модулями системы VisTrails. В системе VisTrails есть единственная иерархия классов модулей и данных. Например, модуль может представлять собой результат вычислений (и, по сути, в каждом модуле есть определяемый по умолчанию выходной порт "self"). Основным недостатком является исчезновение концептуального различия между вычислениями и данными, которые иногда учитываются в системах с архитектурой на базе потоков данных. Но, в этом есть два больших преимущества. Во-первых, точно имитирует систему типов объектов языков Java и C++, и этот выбор был не случайным: для нас важно поддерживать автоматическое подключение больших библиотек классов, таких как VTK. В этих библиотеках допускаются объекты, которые в качестве результата вычислений создают другие объекты, что усложняет такую поддержку в случаях, когда есть различия между вычислениями и данными.

Вторым преимуществом этого решения является то, что становится проще определять значения констант и параметров, устанавливаемых пользователями, и их интеграция с остальной частью системы становится более единообразной. Рассмотрим, например, рабочий процесс, который загружает в сеть локально расположенный файл, заданный как константа. В настоящее время для этого используется графический пользовательский интерфейс, в котором URL может быть указан в качестве параметра (смотрите область редактирования параметров Parameter Edits на рис 23.1). Естественное изменение этого рабочего процесса состоит в том, чтобы можно было определить URL, который уже был вычислен выше. Нам бы хотелось, чтобы остальную часть рабочего процесса требовалось менять как можно меньше. Если предположить, что модули могут выдавать в качестве результата самих себя, то мы бы могли просто подключить строку с правильным значением к порту соответствующего параметра. Поскольку при выдаче константы, результатом ее вычисления является сама эта константа, то поведение было бы точно такое, как если бы значение было фактически определено как константа.

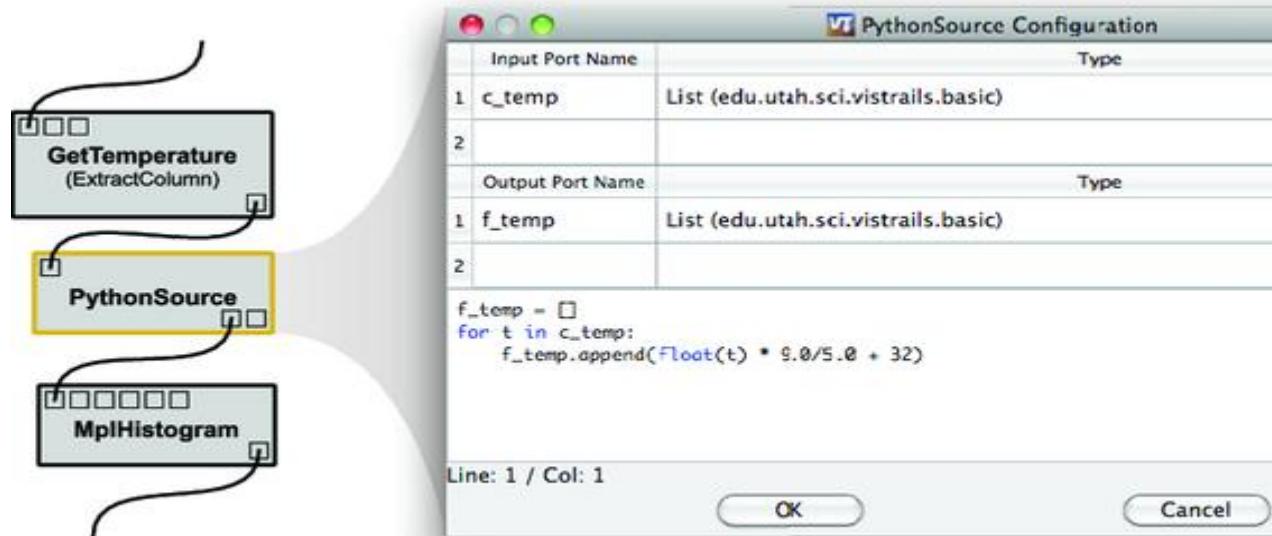


Рис.23.5: Прототипирование новой функциональности с помощью модуля PythonSource

Есть и другие соображения, касающиеся использования констант. Для каждого константного типа есть свой собственный идеальный графический интерфейс для задания значения. Например, в сис-

теме VisTrails в модуле констант — файлов предлагается диалог, позволяющий выбрать файл; значение типа Boolean определяется с помощью отметки, делаемой в чекбоксе; значение цвета выбирается с помощью диалога средства выбора цвета, своего собственного для каждой операционной системы. Чтобы добиться такого обобщения, разработчик должен для конкретной константы создать подкласс базового класса `Constant` и переопределить методы, в которых определяется виджет графического интерфейса и строковое представление (с тем, чтобы любые константы можно было сериализовать на диске).

Отметим, что для простых задач прототипирования, в системе VisTrails имеется встроенный модуль `PythonSource`. Модуль `PythonSource` можно использовать для непосредственной вставки скриптов в рабочий процесс. В конфигурационном окне модуля `PythonSource` (смотрите рис.23.5) предоставляется возможность указать несколько входных и выходных портов, а также указать код на языке Python, который должен быть выполнен.

23.4. Компоненты и возможности

Как уже говорилось выше, в системе VisTrails предоставляется набор функций и пользовательских интерфейсов, упрощающих создание и выполнение исследовательских задач, в которых требуются вычисления. Ниже мы опишем некоторые из них. Мы также кратко обсудим, как система VisTrails используется в качестве инфраструктуры, поддерживающей создание публикаций с большим объемом информации о происхождении данных. Более полное описание системы VisTrails и ее возможностей смотрите в документации, имеющейся в сети [6].

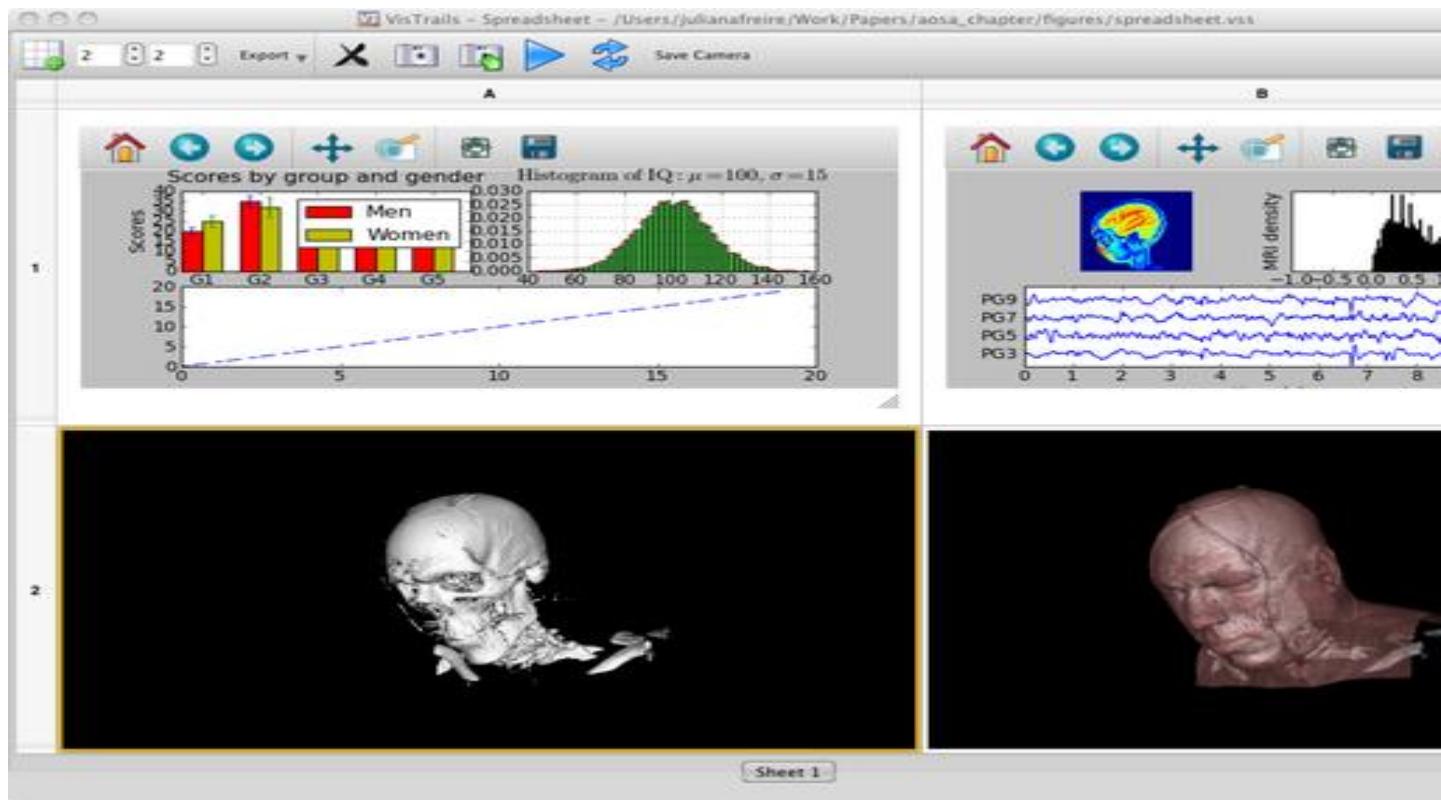


Рис.23.6: Таблица визуализации

23.4.1. Таблица визуализации

Система VisTrails позволяет пользователям с помощью таблицы визуализации исследовать и сравнить результаты, полученные в нескольких рабочих процессах (смотрите рис 23.6). Эта таблица (электронная таблица — прим.пер.) представляет собой пакет VisTrails со своим собственным интерфейсом, состоящий из листов и ячеек. Каждый лист содержит набор ячеек и обладает настраи-

ваемым интерфейсом. Ячейка содержит визуальное представление результатов, созданных рабочим процессом, и ее можно настроить для отображения различных типов данных.

Чтобы отобразить ячейку таблицы, в рабочем процессе должен быть модуль, являющийся производным от базового модуля `SpreadsheetCell`. Каждый модуль `SpreadsheetCell` соответствует отдельной ячейке в таблице, так что в одном рабочем процессе можно создавать несколько ячеек. Метод `compute` модуля `SpreadsheetCell` осуществляет обработку взаимодействия между движком `Execution Engine` (рис. 23.3) и электронной таблицей. Во время выполнения таблица создает ячейку в соответствие с запрошенным ею типом при помощи динамического создания экземпляра класса Python. Таким образом, конкретные визуальные представления можно получить с помощью создания подкласса `SpreadsheetCell`, имеющего метод `compute`, который отправит в таблицу сообщение о конкретном типе ячейки. Например, в рабочем процессе на рис. 23.1, `MplFigureCell` является модулем `SpreadsheetCell`, созданным для показа изображений, созданных с помощью `matplotlib`.

Поскольку в таблице в качестве основы графического пользовательского интерфейса применяется PyQt, виджеты конкретных ячеек должны быть подклассами класса `QWidget` из PyQt. В них также должен быть определен метод `updateContents`, который, когда поступают новые данные, вызывается таблицей для обновления виджета. В виджете каждой ячейки можно с помощью реализации метода `toolbar` дополнительно определить специальную инструментальную панель; когда ячейка выбирается, эта панель будет отображаться на месте инструментальной панели таблицы.

На рис. 23.6 показана таблица для случая, когда выбрана ячейка VTK; в этом случае в инструментальной панели представлены конкретные виджеты, предназначенные для экспорта изображений PDF, возвращать обратно в рабочий процесс информацию о положении камеры и создавать анимации. В пакете электронной таблицы определен настраиваемый виджет `QCellWidget`, в котором предлагаются обычные функции, такие как воспроизведение истории (анимация) и отработка мульти-сенсорных событий. Его можно использовать вместо виджета `QWidget` для более быстрой разработки ячеек новых типов.

Даже хотя в таблице в качестве типов используются виджеты PyQt, можно интегрировать виджеты, написанные с помощью других инструментальных средств, предназначенных для создания графических пользовательских интерфейсов. Чтобы это сделать, виджет должен экспортировать свои элементы на нативную платформу, а затем можно воспользоваться PyQt с тем, чтобы их получить. Мы используем такой подход для виджета `VTKCell`, т. к. виджен фактически написан на языке C++. Во время выполнения, виджет `VTKCell` получает идентификатор обработчика окна Win32, X11 или Cocoa/Carbon в зависимости от используемой системы, и отображает его в канвас таблицы.

Листы могут настраиваться точно также, как и ячейки. По умолчанию, каждый лист имеет табличную компоновку и помещается в окно с закладками. Однако, любой лист можно отсоединить от окна электронной таблицы, чтобы можно было видеть одновременно несколько листов. Также можно с помощью подкласса класса `StandardWidgetSheet`, который является виджетом PyQt, создать другую компоновку листа. Подкласс `StandardWidgetSheet` управляет расположением ячеек, а также взаимодействует с ними в режиме редактирования. В режиме редактирования, пользователи могут манипулировать с расположением ячеек и выполнять дополнительные действия с самими ячейками, а не только с их содержимым. К числу таких действий относится применение аналогий (смотрите раздел 23.4) и создание новых версий рабочих процессов, работающих с исследуемыми параметрами.

23.4.2. Визуальные различия и аналогии

Когда мы разработали систему VisTrails, мы, в добавок, к возможности сбора данных, хотели использовать информацию о происхождении данных. Во-первых, мы хотели, чтобы пользователи

видели точные различия между версиями, но потом мы поняли, что более полезной функцией была бы возможность определять различия в других рабочих процессах. Решение обеих этих задач, возможно, поскольку в системе VisTrails отслеживается эволюция рабочих процессов.

Поскольку в дереве версий собираются все изменения и мы можем инвертировать каждое действие, мы можем найти полную последовательность действий, с помощью которых одна версия преобразуется в другую. Обратите внимание, что некоторые изменения компенсируют друг друга, что делает возможным сократить эту последовательность. Например, нет необходимости при определении различий учитывать добавление модуля, который был позже удален. Наконец, у нас есть некоторые эвристики для дальнейшего упрощения последовательности: когда один и тот же модуль появился в обоих рабочих процессах, но был добавлен с помощью отдельных действий, мы, мы отменить его добавление и удаление.

Анализируя набор изменений, мы можем создать визуальное представление, в котором показаны похожие и различные модули, соединения и параметры. Это проиллюстрировано на рис.23.4. Модули и соединения, которые появляются в обоих рабочих процессах, показаны серым цветом, а те, которые появляются только в одном рабочем процессе, окрашены в тот цвет, который соответствует тому рабочему процессу, в котором они находятся. Совпадающие модули с различными параметрами оттеняются светло серым и пользователь может для каждого модуля проверить различия параметров по таблице, в которой показаны значения для каждого рабочего процесса.

Операция аналогии позволяет пользователям определить эти различия и применить их к другим рабочим процессам. Если пользователь внес ряд изменений в существующий рабочий процесс (например, изменяет разрешение и формат файла для выдаваемого изображения), он может с помощью аналогии применить те же самые изменения к другим рабочим процессам. Для этого пользователь выбирает исходный и целевой рабочие процессы, по которым определяется набор необходимых изменений, а также рабочий процесс, к которому по аналогии должны быть применены эти различия. Система VisTrails вычисляет различие между первыми двумя рабочими процессами, взятыми в качестве шаблона, а затем определяет, как нужно переопределить это различие с тем, чтобы применить его к третьему рабочему процессу. Поскольку различия можно применять к рабочим процессам, которые не совпадают с начальным рабочим процессом, нам нужно нестрогое соответствие (*soft matching*), которое позволяет считать соответствующими аналогичные модули. С помощью такого соответствия, мы можем переопределить различие таким образом, чтобы к выбранному рабочему процессу можно было применить последовательность изменений [SVK +07]. Метод не является абсолютно надежным, и могут создаваться такие новые рабочие процессы, которые будут не совсем такими, как нам хотелось. В таких случаях, пользователь может попытаться исправить любые возникшие ошибки, либо может вернуться к предыдущей версии и применить изменения вручную.

Чтобы вычислить нестрогое соответствие, используемое в аналогиях, мы потребуется соблюсти баланс между локальными соответствиями (одинаковых или очень похожих модулей) с общей структурой рабочего процесса. Отметим, что вычисление даже идентичного соответствия неэффективно из-за строгости изоморфизма подграфов, поэтому мы нам требуется использовать эвристику. Короче говоря, если в двух рабочих процессах два в некотором смысле похожих модуля находятся среди сходных соседей, мы можем сделать вывод, что эти два модуля функционируют аналогичным образом и должны также расцениваться как соответствующие друг другу. Более формально, мы строим продукционный граф, в котором каждый узел является возможным сопряжением модулей в исходных рабочих процессах, а ребра обозначают общие соединения. Затем, мы запускаем шаги рассеяния оценок в каждом узле по ребрам, идущим к соседних узлам. Это марковский процесс, аналогичный ранжированию страниц, используемому Google, который, в конце концов, сойдется, и, в сущности, останется набор оценок, в которых теперь будет некоторая глобальная информация. Исходя из этих оценок мы можем с помощью порога, определяющие очень разнородные модули как непарные, определить лучшее соответствие.

23.4. Компоненты и возможности

23.4.3. Запрос информации о происхождении

Информация о происхождении данных, собираемая системой VisTrails, включает в себя набор, состоящий из процессов, каждый со своей собственной структурой, метаданных и журнала выполнения. Важно то, что пользователи могут получить доступ к этим данным и могут их изучить. В системе VisTrails есть как текстовый, так и визуальный (WYSIWYG) интерфейсы запросов. Для такой информации, как теги, аннотации и даты, пользователь может использовать поиск по ключевым словам с возможностью разметки. Например, найти все рабочие процессы с помощью ключевого слова `plot`, которые были созданы пользователем `user:~dakoop`. При этом запросы о конкретных подграфах рабочего процесса проще представить с помощью визуального интерфейса запросов по образцу, в котором пользователи могут либо создать запрос с нуля, либо скопировать и модифицировать часть уже существующего конвейера.

При разработке такого интерфейса запросов по образцу, мы взяли из существующего редактора рабочих процессов Workflow Editor большую часть кода без изменений, добавив лишь небольшие изменения, параметризующие конструкцию. Что касается параметров, то чаще полезнее искать диапазоны значений и ключевые слова, а не точные значения. Поэтому, в поля значений параметров мы добавили модификаторы; когда пользователи добавляют или изменяют значение параметра, они могут выбрать один из этих модификаторов, которые по умолчанию, задают точное совпадение. Кроме того, что можно сделать визуальный запрос, результат запроса также можно получить в визуальном виде. Совпадающие версии подсвечиваются в дереве версий и любой выбранный рабочий процесс отображается с выделением соответствующей части. Пользователь может выйти из режима результатов запроса инициировав другой запрос или щелкнув по кнопке сброса.

23.4.4. Хранение данных

Система VisTrails хранит информацию о том, как были получены результаты, и с помощью каких шагов. Однако, заново воспроизвести рабочий процесс может быть трудно в случае, если данных, необходимых для рабочего процесса, больше нет. Кроме того, для длительных процессов для того, чтобы избежать повторных перевычислений, может быть полезным между сессиями сохранять промежуточные данные в кэше.

В многих системах, использующих рабочие процессы, в качестве информации о происхождении данных запоминаются пути к файлам в файловой системе, но при таком подходе возможны проблемы. Пользователь может переименовать файл, переместить рабочий процесс в другую систему и не скопировать данные или изменить содержимое данных. В любом из этих случаев, недостаточно в качестве информации о происхождении данных запомнить путь к файлу. Хеширование данных и сохранение хэш значения в информации о происхождении помогут определить, были ли изменены данные, но не помогут найти данные, если они есть. Чтобы решить эту проблему, мы создали пакет Persistence Package - пакет системы VisTrails, использующий инфраструктуру управления версиями для хранения данных, к которым можно обращаться как к информации о происхождении. В настоящее время мы для управления данными используем систему Git, хотя также просто можно пользоваться другими системами.

Мы, чтобы идентифицировать данные, используем универсальные уникальные идентификаторы (UUID) и хэш-значения, получаемые в Git при подтверждении сохранения версий. Если между двумя вычислениями данные изменились, то в репозитарий будет помещена новая версия данных. Таким образом, в любом случае данные можно найти с помощью составного идентификатора - кортежа (`uuid, version`). Кроме того, мы сохраняем хеш-значение и сигнатуру той части рабочего процесса, которая находится выше и с помощью которой были созданы эти данные (если они не являются входными). Это позволяет обращаться к данным, которые могут идентифицироваться

по-разному, а также повторно использовать данные, когда снова запускается то же самое вычисление.

Главным, с чем мы столкнулись при разработке этого пакета было то, как пользователи смогут находить и повторно использовать свои данные. Также нам хотелось, чтобы данные хранились в одном и том же репозитории независимо от того, используются ли они как входные, выходные и промежуточные (выходные данные одного рабочего процесса могут быть входными данными другого рабочего процесса). Есть два основных способа, с помощью которых пользователь может идентифицировать данные: создать новую ссылку на данные или использовать существующую. Обратите внимание, что после первого вычисления новая ссылка станет существующей, поскольку во время исполнения она будет сохранена; позже пользователь, если захочет, может решить создать другую ссылку, но это редкий случай. Поскольку пользователи часто хотят всегда пользоваться последней версией данных, ссылки, задаваемая без указания конкретной версии, будут, по умолчанию, указывать последнюю версию.

Вспомним, что перед выполнением модуля, мы рекурсивно обновляем все его входные данные. Модуль с хранящимися данными не будет обновлять свои входные данные в случае, если вычисления, находящиеся выше, уже отработали. Чтобы это определить, мы берем в репозитории сигнатуру той части рабочего процесса, которая находится выше, и, если такая сигнатура существует, ищем в репозитории ранее вычисленные данные. Кроме того, мы записываем идентификаторы и версии данных в виде информации об их происхождении, с тем, чтобы можно было повторно воспроизвести конкретное вычисление.

23.4.5. Обновления

Поскольку информация о происхождении данных является основой системы VisTrails, ключевым вопросом становится возможность обновлять старые рабочие процессы так, чтобы они могли выполняться с новыми версиями пакетов. Поскольку пакеты могут создаваться сторонними разработчиками, нам необходима как инфраструктура, позволяющая обновлять рабочие процессы, так и специальные средства с тем, чтобы разработчики пакетов могли указывать обновления путей к данным. Основным действием, выполняемым при обновлении рабочего процесса, является замена некоторого модуля его новой версией. Обратите внимание, что это действие осложняется тем, что мы должны изменить все подключения и все параметры старого модуля. Кроме того, например, при изменении интерфейса модуля, обновления могут потребовать переконфигурировать, переназначить или переименовать эти параметры или подключения такого модуля.

Каждый пакет (вместе со связанными с ним модулями) помечается номером версии, и когда эта версия изменяется, мы допускаем, что в этом пакете могут быть изменены также и модули. Обратите внимание, что некоторые из модулей или даже большая их часть могут остаться прежними, но без нашего собственного анализа кода мы это проверить не сможем. Но мы пытаемся автоматически обновить любой модуль, интерфейс которого не изменился. Чтобы сделать это, мы стараемся заменить модуль новой версией и выдать исключительное состояние в случае, если он не работает. Если разработчики изменяют интерфейс модуля или переименовывают модуль, мы даем им возможность указывать эти изменения явно. Чтобы сделать все это более контролируемым, мы создали метод `remap_module`, который позволяет разработчикам указывать только те места, где обновления, выполняемые по умолчанию, должны быть изменены. Например, разработчик, который переименовал входной порт "file" в "value", может указать это конкретное переназначение с тем, чтобы, когда будет создан новый модуль, любое подключение к порту "file" в старом модуле было бы теперь подключением к порту "value". Ниже показан пример обновления для встроенного модуля VisTrails:

```
def handle_module_upgrade_request(controller, module_id, pipeline):
    module_remap = {'GetItemsFromDirectory':
                    [(None, '1.6', 'Directory',
                      {'dst_port_remap':
                        {'dir': 'value'}})]}
```

```

        'src_port_remap':
            {'itemlist': 'itemList'},
        }],
    }
return UpgradeWorkflowHandler.remap_module(controller, module_id, pipeline,
                                            module_remap)

```

Этот фрагмент кода обновлений рабочих процессов, в которых старый модуль `GetItemsFromDirectory` (любой версии до 1.6) заменяется на модуль `Directory`. Здесь порт `dir` старого модуля отображается в порт `value`, а порт `itemlist` в порт `itemList`.

Любое обновление создает в дереве версий новую версию с тем, чтобы можно было отличать друг от друга и сравнивать исполнения рабочих процессов перед и после обновления. Вполне возможно, что обновление изменит выполнение рабочего процесса (например, если разработчик пакета исправит ошибку), и нам нужно отследить эту информацию о происхождении данных. Отметим, что в старых версиях VisTrails, для этого могло требоваться обновлять каждую версию в дереве. Чтобы избежать путаницы, мы обновляем только те версии, к которым пользователь может переходить. Кроме того, мы предоставляем механизм предпочтения, который позволяет пользователю отложить запоминание любого обновления до тех пор, пока рабочий процесс не будет модифицирован или выполнен; если пользователь просто просматривает такую версию, нет необходимости сохранять обновление.

23.4.6. Совместный доступ и публикация результатов, использующих информации о происхождении

Несмотря на то, что воспроизводимость результатов является краеугольным камнем научного метода, в текущих публикациях, описывающих вычислительные эксперименты, часто нет достаточного объема данных, позволяющих повторять или обобщать представленные результаты. В последнее время возобновляется интерес к публикациям с воспроизводимыми результатами. Основное препятствие к более широкому распространению такой практики в том, что трудно создать сборку данных, в которую бы входили все компоненты (например, данные, код, параметры настроек), необходимые при воспроизведении и проверке результата.

Благодаря сбору подробной информации о происхождении данных, а также многим возможностям, описанным выше, система VisTrails делает более простым этот процесс для вычислительных экспериментов, проводимых в рамках системы. Тем не менее, нужны средства как для ссылок на документы, так и для совместного доступа к информации о происхождении данных.

У нас есть разработанные пакеты VisTrails, которые позволяют представлять в статьях результаты вместе с информацией о происхождении данных. С помощью пакета LaTeX, который мы разработали, пользователи могут добавлять рисунки, относящиеся к рабочим процессам VisTrails. С помощью следующего кода на LaTeX будет создаваться рисунок с результатами выполнения рабочего процесса

```

\begin{figure}[t]
{
\vistrail[wfid=119,buildalways=false]{width=0.9\linewidth}
}
\caption{Visualizing a binary star system simulation. This is an image
that was generated by embedding a workflow directly in the text.}
\label{fig:astrophysics}
\end{figure}

```

Если документ собран с использованием pdflatex, то команда `\vistrail` вызовет скрипт Python с параметрами, полученными в ответ на сообщение XML-RPC, посланное на сервер VisTrails с тем, чтобы выполнить рабочий процесс с идентификатором `id` 119. Этот же скрипт Python с помощью генерации команд LaTeX с гиперссылкой `\includegraphics`, в которых указаны параметры ком-

поновки документа (`width=0.9\linewidth`), выполнит загрузку результатов рабочего процесса с сервера и добавит их в результирующий документ PDF.

Кроме того, результаты, полученные в системе VisTrails, можно включать в веб-страницы, страницы вики, документы Word и презентации PowerPoint. Связь между Microsoft PowerPoint и системой VisTrails осуществляется через компонентную модель Component Object Model (COM) и интерфейс связывания и внедрения объектов Object Linking and Embedding (OLE). Чтобы объект взаимодействовал с PowerPoint, в нем должны быть реализованы, по меньшей мере, интерфейсы `IObject`, `IDataObject` и `IPersistStorage` модели COM. Поскольку для сборки нашего объекта OLE мы используем класс `QAxAggregated` из Qt, который является абстракцией реализации интерфейсов COM, то интерфейсы `IDataObject` и `IPersistStorage` будут с помощью Qt созданы автоматически. Таким образом, мы должны реализовать только интерфейс `IObject`. При этом самый важным будет обращение к `DoVerb`. Оно позволит системе VisTrails реагировать на определенные действия PowerPoint, например, на активацию объектов. В нашей реализации, когда объект VisTrails активируется, мы загружаем приложение VisTrails и позволяем пользователям открывать и взаимодействовать с конвейером, который они хотят вставить. После того, как они закроют систему VisTrails, результат работы конвейера будет показан в PowerPoint. Вместе с объектом OLE также хранится информация и о конвейере.

Чтобы позволить пользователям свободно обмениваться своими результатами вместе с сопутствующей информацией о происхождении данных, мы создали сайт crowdLabs. Сайт crowdLabs [7] является социальным веб-сайтом, на котором набор полезных инструментальных средств объединен с масштабируемой инфраструктурой с тем, чтобы можно было предоставить ученым среду для совместного анализа и визуализации данных. Сайт crowdLabs тесно интегрирован с системой VisTrails. Если пользователь хочет предоставить для общего пользования какие-нибудь результаты, полученные в системе VisTrails, он может непосредственно из системы VisTrails подключиться к серверу crowdLabs для загрузки этой информации. После того, как информация будет загружена, пользователи могут обращаться к рабочим процессам и выполнять их через веб-браузер — эти рабочие процессы выполняются сервером VisTrails, который поддерживает работу crowdLabs. Подробности о том, как система VisTrails используется для создания публикаций с воспроизводимыми данными,смотрите по ссылке <http://www.vistrails.org>.

23.5. Усвоенные уроки

К счастью, еще в 2004 году, когда мы начали размышлять о системе исследования и визуализации данных, в которой должна поддерживаться работа с информацией о происхождении данных, мы еще не представляли, насколько будет сложно и сколько времени потребуется для того, чтобы добраться до той точки, в которой мы сейчас находимся. Если бы мы это представляли, то мы, вероятно, никогда бы не начали проект.

На начальном этапе единственной стратегией, которая работала хорошо, было быстрое прототипирование новых функций и передача их выбранной группе пользователей. Первые отзывы и одобрения, которые мы получили от этих пользователей, сыграли важную роль в продвижении проекта вперед. Без обратной связи с пользователями систему VisTrails было бы невозможно разрабатывать. Если и есть единственный аспект проекта, который нам бы хотелось выделить, это то, что большинство функций в системе были разработаны в качестве прямой реакции на обратную связь с пользователями. Тем не менее, стоит отметить, что то, что во многих ситуациях просили пользователи, было не самым лучшим, в чем они нуждались. Снова и снова мы были вынуждены проектировать и перепроектировать функции с тем, чтобы быть уверенными, что они будут полезными и будут должным образом интегрированы в систему.

Если принять во внимание наш подход, ориентированный на пользователей, то можно было бы ожидать, что каждая возможность, которую мы разработали, будет активно использоваться. К сожалению, этого не произошло. Иногда причиной этого было то, что функция была весьма "не-

обычной", поскольку ее нельзя было найти в других инструментальных средствах. Например, аналогии и даже дерево версий не были теми понятиями, с которыми знакомо большинство пользователей и из-за этого им требовалось некоторое время с тем, чтобы с ними освоиться. Другим важным вопросом была документация, вернее отсутствие таковой. Как и во многих других проектах с открытым кодом, мы были намного больше преуспели в разработке новых функций, чем в документировании существующих. Такое отставание в документации ведет не только к неполному использованию полезных функций, но и к многочисленным вопросам в наших списках рассылок.

Одна из проблем изучения такой системы, как VisTrails, является то, что она весьма общая. Несмотря на все наши усилия по улучшению удобства использования, система VisTrails является сложным инструментом и некоторым пользователям требуются значительные усилия на ее изучение. Мы считаем, что с течением времени, с улучшенной документацией, с дальнейшими улучшениями в системе, и с большим количеством примеров ее использования в конкретных областях, усилия по ее освоению станут существенно меньшими. Также, когда концепция информации о происхождении данных станет более распространенной, пользователям будет проще понять философию, которой мы следовали при разработке системы VisTrails.

23.5.1. Благодарности

Мы хотели бы поблагодарить всех талантливых разработчиков, которые внесли свой вклад в систему VisTrails: Erik Anderson, Louis Bavoil, Clifton Brooks, Jason Callahan, Steve Callahan, Lorena Carlo, Lauro Lins, Tommy Ellkvist, Phillip Mates, Daniel Rees и Nathan Smith. Отдельное спасибо Antonio Baptista, который сыграл важную роль, помогая нам развивать концепцию проекта; и Matthias Troyer, чье сотрудничество помогло нам улучшить систему, и, в частности, стало импульсом к разработке и реализации функций, предназначенных для публикации результатов исследований с большим объемом информации о происхождении данных. Исследование и разработка системы VisTrails финансируется Национальным научным фондом по грантам IIS-1050422, IIS-0905385, 0844572 IIS, ATM-0835821, IIS-0844546, IIS-0746500, CNS-0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096, CCF-0401498, OISE-0405402, CCF-0528201, CNS-0551724, а также грантам Министерствам энергетики SciDAC (центры VACET и SDM) и факультета IBM.

Примечания

1. <http://www.vistrails.org>
2. <http://www.hibernate.org>
3. <http://www.sqlobject.org>
4. <http://www.makotemplates.org>
5. <http://www.vtk.org>
6. <http://www.vistrails.org/usersguide>
7. <http://www.crowdlabs.org>

24. Система VTK

Visualization Toolkit (VTK) является широко используемой системой программного обеспечения для обработки и визуализации данных. Она используется в научных вычислениях, анализе медицинских изображений, вычислительной геометрии, рендеринге, обработке изображений и в информатике. В этой главе мы приводим краткий обзор VTK, в том числе некоторые из основных шаблонов проектирования, которые делают VTK успешной системой.

Чтобы действительно разобраться с системой программного обеспечения, важно не только понять, какие проблемы она решает, но также и оценить определенный пласт культуры, в котором она возникла. В случае VTK программное обеспечение было якобы разработано как система 3D визуализации научных данных. Но культурный контекст, в котором оно возникло, добавляет еще доста-

точно длинную предысторию к его существованию, а также помогает объяснить, почему программное обеспечение было разработано и развернуто именно так, как это было сделано.

В те времена, когда система VTK была задумана и написана, ее первые авторы (Уилл Шредер - Will Schroeder, Кен Мартин - Ken Martin и Билл Лоренсен - Bill Lorensen) были исследователями в научно исследовательском отделении GE Corporate R&D. Мы активно вкладывали свои усилия предшествующую систему, известную как LYMB, который была реализована на языке C в среде, похожей на среду Smalltalk. Хотя в то время это была отличная система, мы, как исследователи, при попытке продвижения нашей работы были разочарованы возникновением двух основными препятствий: 1) проблемой лицензирования и 2) тем, что использовалось нестандартное, проприетарное программное обеспечение. Лицензирование стало проблемой из-за попыток распространять программное обеспечение за пределами GE, что было почти невозможно из-за необходимости привлечения только корпоративных юристов. Во-вторых, даже если бы мы были развертывали программное обеспечение внутри GE, многие из наших клиентов отказались бы от специального изучения проприетарной нестандартной системы, поскольку усилия, затрачиваемые работником на ее освоение, становились напрасными как только работник покидал компанию, а также поскольку в системе не было поддержки широко распространенного стандартного набора инструментальных средств. Поэтому, в конечном итоге, основной мотивацией для создания VTK была разработка открытого стандарта, или платформы для совместной работы, с помощью которой мы могли бы легко передавать новые технологии нашим клиентам. Таким образом, выбор для системы VTK лицензии с открытым исходным кодом был, вероятно, самым важным проектным решением, которое мы приняли.

Образцовым решением, как потом оказалось, был окончательный выбор, сделанный авторами, безвозмездной разрешительной лицензии (то есть, лицензии BSD, а не лицензии GPL) и, в конечном счете, позволило оказывать услуги и консалтинг на платной основе, что потом стало деятельностью фирмы Kitware. Тогда, когда мы принимали решение, нас больше всего интересовало уменьшение барьеров для сотрудничества с учеными, научно-исследовательскими лабораториями и коммерческими структурами. С того момента мы обнаружили, что во многих организациях избегают пользоваться безвозмездными лицензиями из-за того потенциального ущерба, который они могут нанести. На самом деле мы утверждаем, что безвозмездные лицензии делают многое для того, чтобы замедлить использование программного обеспечения с открытым исходным кодом, но, это аргументом для другой ситуации. Дело в следующем: одно из основных проектных решений, касающееся любых систем программного обеспечения, является выбор авторских лицензий. Важно рассмотреть цели проекта, а затем соответствующим образом решить проблему лицензирования.

24.1. Что такое VTK?

Система VTK была первоначально задумана как система визуализации научных данных. Многие из тех, кто не связан с вопросами визуализации, наивно считают визуализацию некоторым видом геометрического рендеринга: изучение виртуальных объектов и взаимодействия с ними. Хотя это действительно часть визуализации, в общем визуализация данных включает в себя весь процесс преобразования данных в сенсорное представление, обычно - в изображения, но также включает в себя тактильные, слуховые и другие виды представлений. Формируемые данные содержат не только геометрические и топологические конструкции, в том числе и такие абстракции, как разложение на сеточные или другие сложные пространственные элементы, но атрибуты основной структуры, например, скалярные значения (например, температуру или давление), векторы (например, скорость), тензоры (например, напряжений и деформаций), а также такие атрибуты рендеринга, как нормали к поверхности и текстурные координаты.

Следует отметить, что данные, представляющие пространственно-временную информацию рассматривается, как правило, как часть научной визуализации. Однако есть более абстрактные формы данных, такие как демографические данные по рынкам, веб-страницы, документы и другая

информация, которая может быть представлена только через абстрактные (т.е. не пространственные-временные) отношения, такие как неструктурированные документы, таблицы, графики и деревья. Эти абстрактные данные, как правило, обрабатываются с помощью методов, относящихся к визуализации информации. Благодаря сообществу, система VTK теперь способна обрабатывать научные данные и осуществлять визуализацию информации.

В качестве системы визуализации, роль VTK состоит в получении данных в этой форме и превращении их, в конечном итоге, в формы, которые воспринимаемые сенсорным аппаратом человека. При этом одним из основных требований системы VTK является ее способность создавать конвейеры с потоками данных, через которые можно пропустить, обработать, представить и в конечном счете визуально отобразить данные. Поэтому этот инструментальный набор должен был с архитектурной точки зрения представлять собой гибкую систему и это было отражено на многих уровнях проекта. Например, мы разработали систему специального назначения VTK в виде инструментального средства со многими сменными компонентами, которые можно объединять для обработки широкого спектра данных.

24.2. Архитектурные особенности

Перед тем, как погружаться в конкретных особенности архитектуры VTK, давайте взглянем на высокоуровневые концепции, которые существенно повлияли на разработку и использование системы. Одной из таких особенностей является возможность в VTK использовать гибридные обертки. С помощью этой возможности в реализации C++ в VTK автоматически создаются привязки к языкам Python, Java и Tcl (можно добавить и были добавлены дополнительные языки). Наиболее опытные разработчики будут работать в языке C++. Пользователи и разработчики приложений могут использовать язык C++, но часто для них предпочтительнее интерпретируемые языки, упомянутые выше. Этот гибрид среди компиляции/интерпретации сочетает в себе лучшее из двух миров: высокую производительность ресурсоемких алгоритмов и гибкость при прототипировании или разработке приложений. На самом деле в сообществе научных вычислений многим нравится такой подход к многоязычным вычислениям и они часто пользуются системой VTK в качестве шаблона для разработки своего собственного программного обеспечения.

Если рассматривать процесс разработки программного обеспечения, то в системе VTK используется CMake для управления сборкой, CDash/CTest - для тестирования и CPack - для кросс-платформенного развертывания пакетов. Действительно, система VTK может быть откомпилирована практически на любом компьютере, в том числе и на суперкомпьютерах, которые зачастую обладают заведомо скромными средами разработки. Кроме того, есть средства генерации веб-страниц, вики, списков рассылок (для пользователей и для разработчиков) и документации (то есть, Doxygen), а инструментальные средства дополнены треккером ошибок (Mantis).

24.21. Основные особенности

Поскольку VTK является объектно-ориентированной системой, в VTK тщательно контролируется доступ к элементам класса и экземплярам данных. Обычно все элементы данных являются либо защищенными (т. е. protected), либо приватными (т. е. private). Доступ к ним осуществляется через методы Set и Get с особыми вариациями, относящимися к логическим данным, модальным данным, строкам и векторам. Многие из этих методов на самом деле создается с помощью макросов, вставленных в заголовочные файлы класса. Так, например:

```
vtkSetMacro(Tolerance,double);
vtkGetMacro(Tolerance,double);
```

после раскрытия будут выглядеть следующим образом:

```
virtual void SetTolerance(double);
virtual double GetTolerance();
```

Есть много причин, выходящих за рамки простого улучшения ясности кода, для использования этих макросов. В VTK существуют важные элементы данных, управляющие процессом отладки, обновляющие значения времени изменения объектов (MTime), а также надлежащим образом управляющие подсчетом ссылок на объекты. Эти макросы обрабатывают такие данные правильным образом и использование этих макросов настоятельно рекомендуется. Например, особенно пагубная ошибка в системе VTK происходит тогда, когда с помощью Mtime не удается правильно управлять объектами. В этом случае код может выполняться не тогда, когда это надо, или может выполнять слишком часто.

Одной из сильных сторон системы VTK является ее сравнительно простые средства представления и управления данными. Обычно для представления подряд идущих частей информации используются различные массивы данных определенного типа (например, `vtkFloatArray`). Например, список из трех точек XYZ может быть представлен в `vtkFloatArray` девятью записями (x,y,z, x,y,z и т. д.). Для таких массивов существует понятие кортежа, поэтому 3D-точка представляет собой кортеж из трех элементов, тогда как симметричная тензорная матрица размером 3×3 представляет собой кортеж, состоящий из 6 элементов (в некоторых из которых можно, благодаря наличию в них симметрии, сэкономить место). Такая архитектура была взята за основу специально, поскольку в научных вычислениях она обычно используется в качестве интерфейса с системами, обрабатывающими массивы (например, Fortran), и с ее помощью гораздо эффективнее выделять и освобождать память в больших фрагментах данных, следующих непрерывно друг за другом. Кроме того, взаимодействие с данными, сериализация и выполнение операций ввода-вывода, как правило, происходит гораздо эффективнее, когда данные непрерывно следуют друг за другом. В системе VTK с помощью этих основных массивов данных (различных типов) происходит представление многих данных и для них есть много разнообразных удобных методов, используемых для добавления информации и для доступа к ней, в том числе методы для быстрого доступа, и методы, которые при добавлении новых данных могут, по мере необходимости, автоматически выделять память. Массивы данных являются подклассами абстрактного класса `vtkDataArray`, что означает, что для упрощения кодирования могут быть использованы общие виртуальные методы. Однако, для более высокой производительности используются статические шаблонные функции, в которых в зависимости от типа данных происходит переключение с последующим прямым доступом к массивам с непрерывно следующими данными.

Обычно шаблоны C++ не видны в интерфейсе API с общедоступными классами, хотя они широко используются с целью повышения производительности. Это также касается STL: для того, чтобы скрыть сложность реализации шаблона от пользователей и разработчиков приложений, мы обычно используем шаблон разработки PIMPL [1]. Он нам особенно полезен в случае, когда дело доходит до обверток вокруг кода в интерпретируемом коде так, как это описано выше. Поскольку в общедоступном коде API удается избегать использовать сложные шаблоны, то это означает, что, с точки разработчиков приложений, реализация системы VTK такова, чтоней не сложно выбирать типы данных. Конечно, если заглянуть глубже, то исполнение кода происходит с использованием тех типов данных, которые определяются на этапе выполнения программы, когда собственно и происходит к ним доступ.

Некоторые пользователи задаются вопросом, почему в системе VTK для управления памятью используется подсчет ссылок на данные, а не используется более удобный для пользователей подход, например, сборка мусора. Основной ответ состоит в том, что поскольку размеры данных могут быть огромными, в VTK нужен полный контроль над тем, что происходит при удалении данных. Например, объем матрицы данных размером $1000 \times 1000 \times 1000$, размер каждого элемента в которой равен одному байту, будет составлять гигабайт данных. Достаточно неразумно ждать того момента, пока сборщик мусора решит, нужно или не нужно освобождать память от этой матрицы. В большинстве классов (подклассов класса `vtkObject`) в системе VTK имеется встроенная возможность подсчета ссылок. В каждом объекте есть счетчик ссылок, который инициализируется единицей, когда создается экземпляр объекта. Каждый раз, когда объект регистрируется, значение счетчика увеличивается на единицу. Аналогичным образом, когда выполняется операция, обратная регистрации (или, что эквивалентно тому, что объект удаляется), счетчик ссылок уменьшается

на единицу. В конце концов счетчик ссылок на объект становится равным нулю и в этот момент происходит самоуничтожение объекта. Типичный пример выглядит следующим образом:

```

vtkCamera *camera = vtkCamera::New();           // счетчик ссылок равен 1
camera->Register(this);                      // счетчик ссылок равен 2
camera->Unregister(this);                     // счетчик ссылок равен 1
renderer->SetActiveCamera(camera);           // счетчик ссылок равен 2
renderer->Delete();                          // счетчик ссылок равен 1 когда renderer
удаляется is deleted
camera->Delete();                           // camera самоуничтожается

```

Есть еще одна важная причина, почему в системе VTK важно подсчитывать число ссылок: предоставляется возможность эффективного копирования данных. Например, представьте себе объект данных D1, в котором находится несколько массивов данных: точки, полигоны, цвета, скаляры и текстурные координаты. Теперь представьте обработку этих данных, когда создается новый объект данных D2, который является таким же, как первый, плюс добавляются векторные данные (расположенные в точках). Расточительным подходом является полное (со всеми элементами) копирование объекта D1 в создаваемый объект D2, а затем добавление к D2 нового массива векторов данных. Либо мы создаем пустой объект D2, а затем передаем массивы из D1 в D2 (поверхностное копирование), используя подсчет ссылок для отслеживания владельца данных, и, наконец, добавляем к D2 новый массив векторов. (*Прим.пер.: здесь, скорее всего идет речь о том, что создаются ссылки на массивы, а не копируются сами массивы*). Последний подход позволяет избежать копирования данных, что, как мы показали ранее, имеет важное значение для хорошей системы визуализации. Как мы увидим далее в этой главе, конвейер обработки данных постоянно выполняет операции такого рода, т. е. копирует данные из входных данных алгоритма в выходные данные и, следовательно, выполняет подсчет ссылок на данные, что необходимо в системе VTK.

Конечно, есть несколько проблем, связанных с подсчетом ссылок. Иногда могут существовать циклические ссылки, когда объекты во взаимодополняющей конфигурации ссылаются друг на друга в цикле. В этом случае требуется интеллектуальное вмешательство, или, как в случае VTK, используется специальный механизм, реализованный в `vtkGarbageCollector`, с помощью которого происходит управление объектами, задействованными в циклах. Когда обнаруживается такой класс (предполагается, что это происходит во время разработки), класс самостоятельно регистрируется в сборщике мусора и выполняет перезагрузку своих собственных методов `Register` и `UnRegister`. Затем, при последующем удалении объекта (или выполнении операции, обратной регистрации) метод выполняет топологический анализ в сетке подсчета ссылок и ищет автономные островки из объектов, ссылающихся друг на друга. Затем эти объекты удаляются сборщиком мусора.

Большинство экземпляров в системе VTK строится через фабрику объектов, реализованную как статический элемент класса. Синтаксис типичного примера выглядит следующим образом:

```
vtkLight *a = vtkLight::New();
```

Здесь важно признать, что, в действительности, экземпляр может создаваться не классом `vtkLight`, он может создаваться подклассом класса `vtkLight` (например, `vtkOpenGLLight`). Есть целый ряд факторов, влияющих на фабрику объектов, наиболее важными из которых является переносимость приложений и независимость от устройств. Например, в приведенном выше мы создаем свет в сцене, для которой выполняется операция рендеринга. В конкретном приложении на конкретной платформе `vtkLight::New` может в результате создать свет с помощью библиотеки OpenGL, однако на других платформах потенциально возможно, что для создания света в графической системе используются другие библиотеки рендеринга. Именно поэтому производный класс, используемый для создания экземпляра объекта, является функцией, зависящей от системной информации времени выполнения. Сначала в системе VTK было множество вариантов, в том числе gl, PHIGS, Starbase, XGL и OpenGL. Хотя большинство из этих теперь исчезли, но появились новые подходы в том числе DirectX и подходы, базирующиеся на использовании графиче-

ских процессоров. По прошествии времени приложения, написанные с использованием VTK, не должны были измениться поскольку разработчики определили подклассы новых конкретных устройств для класса `vtkLight` и других классов, используемых для рендеринга, которые поддерживают развивающуюся технологию. Другим важным использованием фабрики объектов является возможность на этапе выполнения приложения выполнять замены, повышающие производительность. Например, класс `vtkImageFFT` может быть заменен классом, который получает доступ к аппаратным устройствам специального назначения или к библиотеке численной обработки данных.

24.2.2. Представление данных

Одна из сильных сторон системы VTK является возможность с ее помощью представлять сложные формы данных. Эти формы данных варьируются от простых таблиц до сложных структур, таких как сетки конечных-элементов. Все эти формы данных являются подклассами класса `vtkDataObject` так, как это показано на рис.24.1 (обратите внимание, это частичная диаграмма наследования многие классы объектов данных).

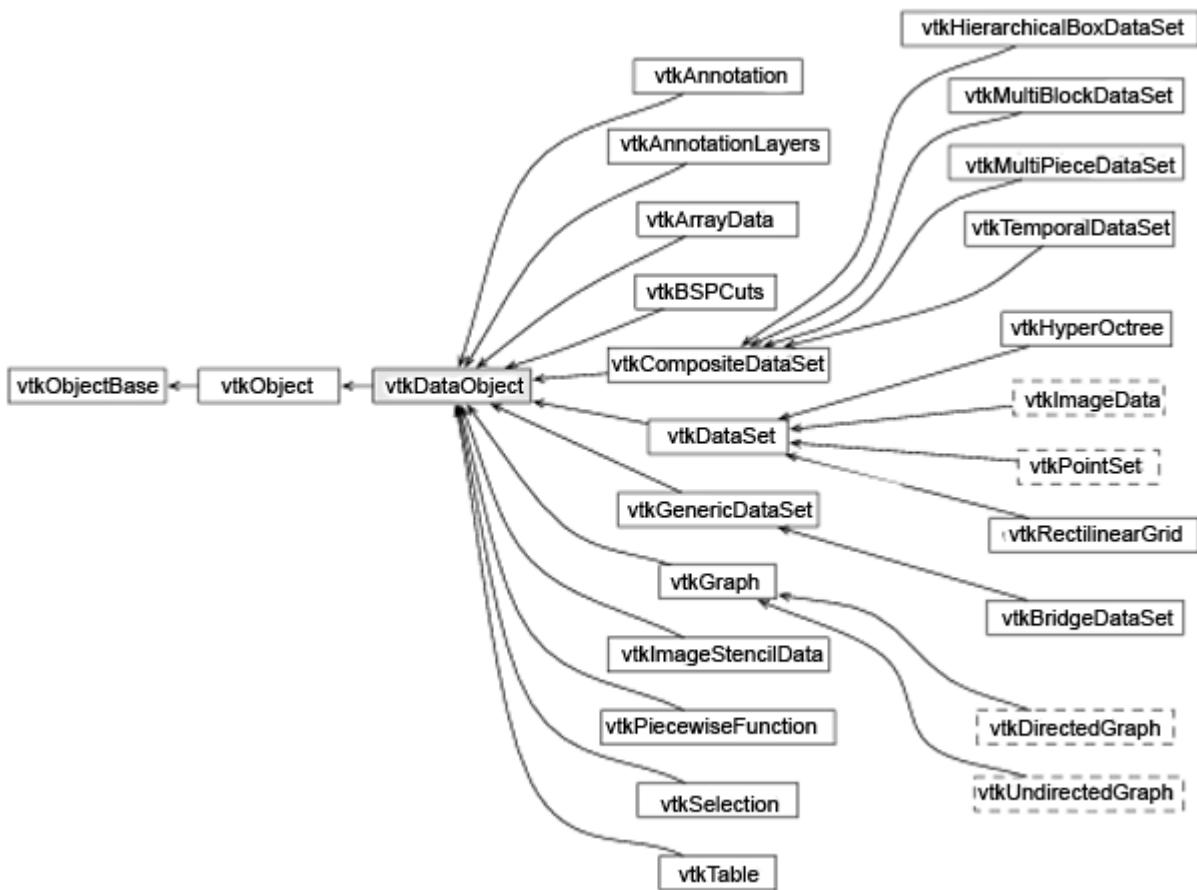


Рис.24.1: Классы объектов данных

Одной из наиболее важных характеристик класса `vtkDataObject` является то, что он может быть обработан в конвейере визуализации (следующий раздел). Среди многих показанных классов, есть только несколько, которые обычно используются в большинстве реальных приложений. Класс `vtkDataSet` и производные классы используются для научной визуализации (рис.24.2). Например, класс `vtkPolyData` используется для представления полигональных сеток; класс `vtkUnstructuredGrid` — для представления сеток, а класс `vtkImageData` - для представления двухмерных и трехмерных пиксельных и воксельных данных.

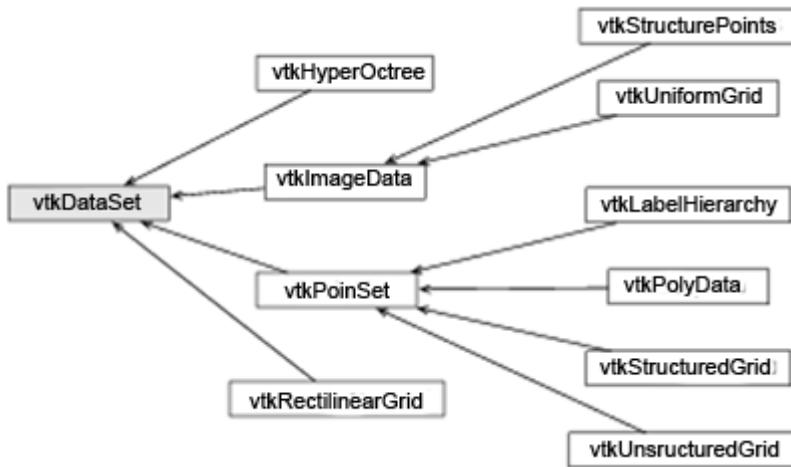


Рис.24.2: Классы наборов данных

24.2.3. Конвейерная архитектура

Система VTK состоит из нескольких основных подсистем. Вероятно, подсистема в большей мере ассоциируется с пакетами визуализации, с помощью которых формируется архитектуру потоков данных/конвейеров. Концептуально конвейерная архитектура состоит из трех основных классов объектов: объектов, используемых для представления данных (объекты класса `vtkDataObject` — смотрите выше), объектов, используемых для обработки, преобразования, фильтрации или отображения объектов данных из одной формы в другую (`vtkAlgorithm`), и объектов, используемых для работы конвейера (`vtkExecutive`), управление которым осуществляется согласно связному графу, в котором чередуются объекты данных и процессов (т.е. конвейеров). На рис.24.3 показан типичный конвейер.

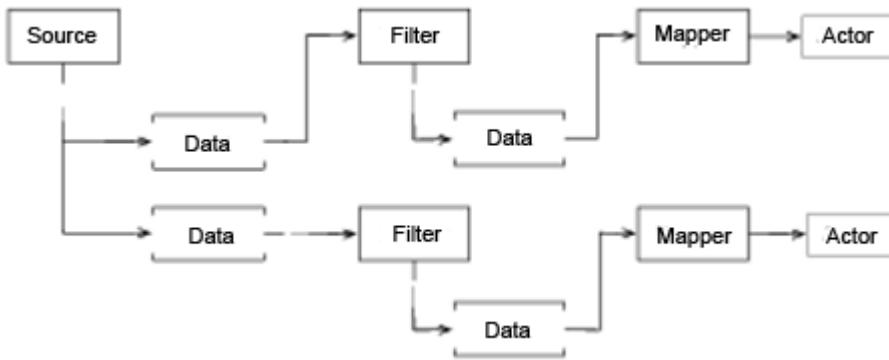


Рис.24.3: Типичный конвейер

Несмотря на концептуальную простоту, действительная реализация конвейерной архитектурой является сложной задачей. Одной из причин этого является то, что представление данных может быть сложным. Например, некоторые наборы данных состоят из иерархии или групп данных, поэтому для обработке всех данных потребуется нетривиальная итерация или рекурсия. Чтобы справиться с составной сложностью с помощью параллельной обработки (независимо от того, касается ли это совместно используемой памяти или масштабируемых подходов распределенной обработки), необходимо разбивать данных на части, причем может потребоваться, чтобы части перекрывались друг с другом с тем, чтобы можно было непрерывно вычислять граничную информацию, например, производные.

Объекты алгоритмов также вносят свои особые сложности. Для некоторых алгоритмов может потребоваться несколько входных потоков и / или они могут создавать несколько потоков выходных данных различных типов. Некоторые из них могут обрабатывать данные локально (например, вычислять центр ячейки), а для других требуется глобальная информация, например, при вычисле-

нии гистограммы. Во всех случаях, алгоритмы воспринимают свои входные данные как неизменяемые, они их только читают для того, чтобы создать свои выходные данные. Это связанно с тем, что данные могут предлагаться в качестве входных данных для нескольких алгоритмов, и будет не очень хорошо, когда один алгоритм будет портить входные данные другого алгоритма.

Наконец, сложность выполнения алгоритма может зависеть от конкретных особенностей стратегии выполнения. В некоторых случаях у нас есть возможность кэшировать промежуточные результаты, получаемые между фильтрами. Это в случае, если в конвейере что-то будет изменено, сведет к минимуму количество повторных расчетов, которые должны быть выполнены. С другой стороны, наборы данных, используемые для визуализации, могут быть огромными, и, в таком случае, мы, возможно, захотим избавиться от данных, если они больше не нужны для вычислений. Наконец, существуют сложные стратегии исполнения, например, обработка данных с несколькими вариантами точности, что требует, чтобы конвейер работал в итеративном режиме.

Чтобы продемонстрировать некоторые из этих концепций и затем объяснять конвейерную архитектуру, рассмотрим следующий пример на языке C++:

```
vtkPExodusIIReader *reader = vtkPExodusIIReader::New();
reader->SetFileName("exampleFile.exo");

vtkContourFilter *cont = vtkContourFilter::New();
cont->SetInputConnection(reader->GetOutputPort());
cont->SetNumberOfContours(1);
cont->SetValue(0, 200);

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(cont->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOutputPort());
writer->SetFileName("outputFile.vtp");
writer->Write();
```

В этом примере объект `reader` читает большой неструктурированный сеточный файл данных (или сетку). Следующий фильтр создает из сетки изоповерхность. Фильтр `vtkQuadricDecimation` уменьшает размер изоповерхности, которая является полигональным набором, уменьшая для этого количество составляющих элементов (т.е., уменьшая количество треугольников, представляющих собой изоконтур). Наконец, после прореживания новый файл с уменьшенным количеством данных записывается обратно на диск. Фактическая работа конвейера происходит тогда, когда объект `writer` вызывает метод `Write` (т.е. в момент запроса данных).

Как показано в этом примере, механизм выполнения конвейеров в системе VTK запускается при запросе данных. Когда некоторому процессу, например, объекту `writer` или `tapper` (объекту, осуществляющему рендеринг данных), требуются данные, он запрашивает их в качестве входа. Если во входном фильтре уже есть соответствующие данные, то фильтр просто возвращает управление выполнением в запрашиваемый процесс. Однако, если на входе нет соответствующих данных, их необходимо вычислить. Следовательно, нужно сначала запросить данных на вход. Этот процесс будет продолжаться в направлении, обратном движению данных по конвейеру, до тех пор, пока не будет достигнут фильтр или источник данных, у которого есть «соответствующие данные», или до тех пор, пока не будет достигнуто начало конвейера, после чего фильтры будут выполнены в правильном порядке, а данные поступят в то место конвейера, где они были запрошены.

Здесь следует пояснить, что означает «соответствующие данные». По умолчанию после того, как в VTK выполняется источник данных или фильтр, его выходные данные помещаются конвейером в кэш с тем, чтобы в будущем избежать ненужных вычислений. Это сделано для того, чтобы минимизировать количество вычислений и/или объем ввода/вывода за счет использования памяти, причем такое поведение является настраиваемым. В конвейере кэшируются не только объекты дан-

ных, но также метаданные об условиях, при которых эти объекты данных были получены. В этих метаданных есть метка времени (т.е. ComputeTime), которая создается в тот момент, когда объект данных был вычислен. Таким образом, в простейшем случае, «соответствующие данные» это такие данные, которые были вычислены после того, как были внесены изменения во все конвейерные объекты, находящиеся по конвейеру раньше конкретного места. Такое поведение проще продемонстрировать с помощью следующих примеров. Давайте в конце предыдущей программы VTK добавим следующие строки:

```
vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
writer2->SetInputConnection(deci->GetOutputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();
```

Как объяснялось ранее, первый вызов `writer->Write` будет причиной того, что произойдет выполнение всего конвейера. Когда вызывается `writer2->Write()`, конвейер, когда он сравнил временну метку кэша со временем изменений прореживающего фильтра (`deci`), контурного фильтра и объекта `reader`, он поймет, что на выходе прореживающего фильтра находятся обновленные данные. Таким образом, запрос данных не должен распространяться ранее, чем до обращения к `writer2`. Теперь, давайте рассмотрим следующие изменения.

```
cont->SetValue(0, 400);

vtkXMLPolyDataWriter *writer2 = vtkXMLPolyDataWriter::New();
writer2->SetInputConnection(deci->GetOutputPort());
writer2->SetFileName("outputFile2.vtp");
writer2->Write();
```

Сейчас при выполнении конвейера будет понятно, что после того, как последний раз были вычислены выходные данные контурного и прореживающего фильтров, контурный фильтр был изменен. Таким образом, кэш для этих двух фильтров устарел, и эти фильтры должны быть вычислены повторно. Однако, поскольку объект `reader`, находящийся перед контурным фильтром, изменен не был, данные, находящиеся в его кэше, остаются действительными, и, следовательно, объект `reader` повторно выполняться не должен.

Сценарий, описанный здесь, является простейшим примером конвейера, выполнение которого осуществляется по запросу. Конвейер системы VTK является гораздо более сложным. Когда фильтру или процессу требуются данные, то может предоставляться дополнительная информация, указывающая конкретные подмножества данных. Например, фильтр может выполнять вспомогательный анализ, запрашивая только часть потока данных. Давайте для демонстрации изменим наш предыдущий пример.

```
vtkXMLPolyDataWriter *writer = vtkXMLPolyDataWriter::New();
writer->SetInputConnection(deci->GetOutputPort());
writer->SetNumberOfPieces(2);

writer->SetWritePiece(0);
writer->SetFileName("outputFile0.vtp");
writer->Write();

writer->SetWritePiece(1);
writer->SetFileName("outputFile1.vtp");
writer->Write();
```

Здесь объект `writer` делает запрос на загрузку и обработку двух частей потока данных, независимых друг от друга, который направляется к началу конвейера. Вы могли заметить, что простая логика выполнения, описанная ранее, здесь работать не будет. По этой логике когда функция `write` вызывается во второй раз, конвейер не должен повторно осуществлять выполнение, т.к. ничего в начале конвейера не изменилось. Поэтому для решения этого более сложного случая, в механизме исполнения закладывается дополнительная логика, позволяющая обрабатывать частей запросов,

таких как этот запрос. В действительности выполнение конвейера в системе VTK состоит из нескольких проходов. Вычисление объектов данных, на самом деле, является последним проходом. Проход, который был выполнен до этого, является запрашивающим проходом. Это тот проход, в котором потребители данных и фильтры могут сообщить в начало конвейера, что им нужно от предстоящего вычисления. В приведенном выше примере объект writer передаст на свой вход, что ему нужно часть с номером 0. Этот запрос будет, на самом деле, передан вплоть до объекта reader. Когда конвейер будет выполняться, reader будет знать, что он должен читать подмножества данных. Кроме того, в метаданных объекта будет запомнена информация о том, какая часть соответствующих данных закэширована. В следующий раз, когда фильтр запрашивает данные со своего входа, эти метаданные будут сравниваться с текущим запросом. Таким образом, в этом примере конвейер будет выполнять повторный пересчет с тем, чтобы обработать запрос других данных.

Есть несколько типов запросов, которые может делать фильтр. К ним относятся запросы конкретного шага по времени, конкретного структурного расширения или количества скрытых слоев (т.е. граничных слоев, необходимых для вычисления информации о соседних данных). Кроме того, во время запрашивающего прохода, каждый фильтр по мере того, как через него проходит запрос, может его изменять. Например, фильтр, который не в состоянии обрабатывать потоки (например, streamline фильтр) может игнорировать частичный запрос и может запросить все данные.

24.2.4. Подсистема рендеринга

На первый взгляд в системе VTK присутствует простая объектно-ориентированная модель визуализации с классами, соответствующими компонентам, с помощью которых создаются трехмерные сцены. Например, объекты `vtkActor` являются объектами, рендеринг которых осуществляется с помощью `vtkRenderer` в сочетании с `vtkCamera` и, возможно, с несколькими объектами `vtkRenderer`, существующими в `vtkRenderWindow`. Сцена освещается одним или несколькими объектами `vtkLight`. Управление положением каждого `vtkActor` происходит с помощью `vtkTransform`, а внешний вид актера определяется через `vtkProperty`. Наконец, геометрическое представление актера определяется с помощью `vtkMapper`. Важную роль в системе VTK играют преобразователи `mapper`, которые обрабатывают завершение обработки данных в конвейере, а также являются интерфейсом системы рендеринга. Рассмотрим следующий пример, в котором мы прореживаем данные и записываем результат в файл, а затем с помощью преобразователя `mapper` визуализируем их и будем с ними взаимодействовать.

```

vtkOBJReader *reader = vtkOBJReader::New();
reader->SetFileName("exampleFile.obj");

vtkTriangleFilter *tri = vtkTriangleFilter::New();
tri->SetInputConnection(reader->GetOutputPort());

vtkQuadricDecimation *deci = vtkQuadricDecimation::New();
deci->SetInputConnection(tri->GetOutputPort());
deci->SetTargetReduction( 0.75 );

vtkPolyDataMapper *mapper = vtkPolyDataMapper::New();
mapper->SetInputConnection(deci->GetOutputPort());

vtkActor *actor = vtkActor::New();
actor->SetMapper(mapper);

vtkRenderer *renderer = vtkRenderer::New();
renderer->AddActor(actor);

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);

vtkRenderWindowInteractor *interactor = vtkRenderWindowInteractor::New();
interactor->SetRenderWindow(renWin);

```

```
renWin->Render();
```

Здесь один актер, механизм рендеринга и окно рендеринга создаются с добавлением преобразователя mapper, который подсоединяет конвейер к системе рендеринга. Также обратите внимание на добавление vtkRenderWindowInteractor, экземпляры которого перехватывают события мыши и клавиатуры и транслируют их в манипуляции с камерой или в другие действия. Этот процесс трансляции определяется через vtkInteractorStyle (подробнее об этом ниже). По умолчанию настройка многих экземпляров объектов и значений данных происходит за кулисами. Например, конструируется идентичное преобразование (identity transform), а также единственный используемый по умолчанию свет (осветитель) и его свойства.

Со временем эта объектная модель стала еще более сложной. Большая часть сложности была привнесена из разработки производных классов, которые специализируются на каком-то одном из аспектов процесса рендеринга. Теперь объекты vtkActor уточняются с помощью vtkProp (подобно тому как свойства prop ищутся на каждой стадии), и есть целая куча этих свойств prop для рендеринга двухмерной графики с оверлеями, текста, специальных трехмерных объектов и даже для поддержки улучшенных методов рендеринга, например, объемного рендеринга или поддержки использования графических процессоров (смотрите рис.24.4).

Поскольку модель данных, поддерживаемая в системе VTK, существенно выросла, аналогичным образом появились различные преобразователи mapper, с помощью которых организуется интерфейс между данными и системой рендеринга. Еще одной областью значительного расширения системы является иерархия трансформаций. То, что первоначально было простой линейной матрицей преобразования размером 4×4 , стало мощной иерархией, в которой поддерживаются нелинейные преобразования, в том числе преобразования сплайнов вида thin-plate. Например, исходный класс vtkPolyDataMapper имел подклассы для конкретных устройств (например, vtkOpenGLPolyDataMapper). В последние годы он был заменен сложным графическим конвейером, называемым painter-конвейером, показанным на рис.24.4.

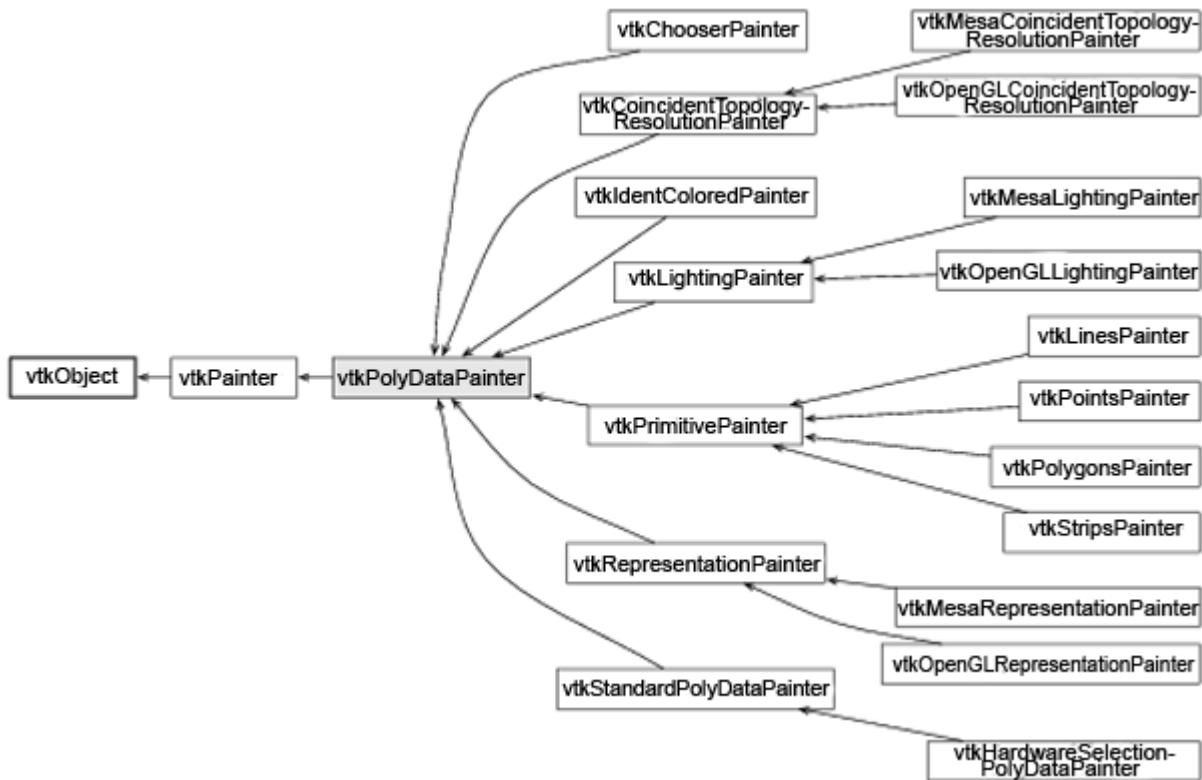


Рис.24.4: Изображение классов

В конструкции painter-конвейера поддерживаются различные методы рендеринга данных, которые можно объединять для получения специальных эффектов. Эта возможность значительно превосходит возможности классического конвейера.

ходит возможности простого преобразователя `vtkPolyDataMapper`, который первоначально был реализован в 1994 году.

Другим важным аспектом системы визуализации является подсистема, позволяющая делать выбор. В системе VTK есть иерархия средств выбора типа `picker`, которые условно делятся на те, что позволяют выбирать свойства `vtkProp` с использованием методов, реализованных аппаратно, а не программно (например, прорисовка лучей); а также на те которые после выполнения операции выбора `pick` позволяют предоставлять информацию с различными уровнями детализации. Например, с помощью некоторых команд `picker` можно получить информацию о месте XYZ в реальном пространстве без указания, какое было выбрано свойство `vtkProp`; тогда как с помощью других команд можно получить не только конкретное свойство `vtkProp`, но конкретную точку или ячейку, из которых состоит сетка, определяющая геометрию свойств `prop`.

24.2.5. События и взаимодействие

Взаимодействие с данными является неотъемлемой частью визуализации. В системе VTK это происходит в различных формах. На самом простейшем уровне, пользователи могут наблюдать события и реагировать на них соответствующим образом с помощью команд (шаблон проектирования «команда/наблюдатель»). Во всех подклассах класса `vtkObject` поддерживаются списки наблюдателей, которые самостоятельно регистрируются в объекте. Во время регистрации, наблюдатели указывают, какое именно событие (или события) их интересует, и добавляют соответствующую команду, которая будет вызвана, если и когда происходит это событие. Чтобы увидеть, как это работает, рассмотрим следующий пример, в котором фильтр (здесь полигональный прореживающий фильтра) имеет наблюдателя, следящим за тремя событиями `StartEvent` (начало события), `ProgressEvent` (продолжение события) и `EndEvent` (завершение события). Эти события вызываются, когда фильтр начинает выполнение, периодически во время выполнения, а затем по окончанию выполнения. Затем в классе `vtkCommand` есть метод `Execute`, который выводит соответствующую информацию, касающуюся того, сколько потребовалось времени для выполнения алгоритма:

```
class vtkProgressCommand : public vtkCommand
{
public:
    static vtkProgressCommand *New() { return new vtkProgressCommand; }
    virtual void Execute(vtkObject *caller, unsigned long, void *callData)
    {
        double progress = *(static_cast<double*>(callData));
        std::cout << "Progress at " << progress << std::endl;
    }
};

vtkCommand* pobserver = vtkProgressCommand::New();

vtkDecimatePro *deci = vtkDecimatePro::New();
deci->SetInputConnection( byu->GetOutputPort() );
deci->SetTargetReduction( 0.75 );
deci->AddObserver( vtkCommand::ProgressEvent, pobserver );
```

Хотя это примитивная форма взаимодействия, она является основополагающей для многих приложений, в которых используется система VTK. Например, простой код, показанный выше, может быть легко преобразован для отображения и управления линейным индикатором процесса, который есть в графическом интерфейсе. Такая подсистема «команда/наблюдатель» также будет центральной в трехмерных виджетах в системе VTK, которые являются сложными интерактивными объектами, используемыми для выполнения запросов, обработки и редактирования данных, и которые описываются ниже.

Как показано в примере, приведенном выше, важно отметить, что события в системе VTK являются предопределенными, но есть скрытая лазейка для событий, которые может определять пользователь. В классе `vtkCommand` определяется набор перечисляемых событий (например,

`vtkCommand::ProgressEvent` в приведенном выше примере), а также событие, определяемое пользователем. Событие `UserEvent`, которое является просто интегрированным значением, обычно используемым в качестве смещения от начального значения в наборе событий, определяемых в приложении пользователем. Так, например, `vtkCommand::UserEvent+100` может относиться к конкретному событию, которое не входит в набор событий, определенных в системе VTK.

С точки зрения пользователя, виджет VTK выступает на сцене в качестве актера, за исключением того, что пользователь может взаимодействовать с ним, манипулируя рукоятками или другими геометрическими элементами (манипуляции рукоятками и геометрическими элементами базируются использовании функций выбора `pick`, которые были описаны ранее). Взаимодействие с таким виджетом в известной степени интуитивно понятно: пользователь берется за сферические рукоятки и перемещает их, или захватывает линию и перемещает ее. Но за кулисами происходит возникновение событий (например, `InteractionEvent`) и приложение, запрограммированное должным образом, может отслеживать эти события, а затем принимать соответствующие меры. Например, при возникновении события `vtkCommand::InteractionEvent` часто происходит следующее:

```
vtkLW2Callback *myCallback = vtkLW2Callback::New();
myCallback->PolyData = seeds;           // streamlines создают точки, обновляемые при взаимодействии
myCallback->Actor = streamline;        // актер streamline становится видимым при взаимодействии

vtkLineWidget2 *lineEditWidget = vtkLineWidget2::New();
lineEditWidget->SetInteractor(iren);
lineEditWidget->SetRepresentation(rep);
lineEditWidget->AddObserver(vtkCommand::InteractionEvent, myCallback);
```

Виджеты VTK фактически построены с использованием двух объектов: подкласса класса `vtkInteractorObserver` и подкласса класса `vtkProp`. Наблюдатель `vtkInteractorObserver` просто наблюдает в окне рендеринга за взаимодействием с пользователем (т.е. за событиями мыши и клавиатуры) и обрабатывает их. Манипулирование подклассами класса `vtkProp` (то есть, актеры) просто происходит с помощью `vtkInteractorObserver`. Обычно такая манипуляция включает изменение геометрии `vtkProp`, к которой относятся управление освещенностью, изменение вида курсора и/или преобразование данных. Конечно, конкретные особенности виджетов требуют, чтобы были написаны подклассы, позволяющие управлять нюансами его поведения, и в настоящее время в системе есть более 50 различных виджетов.

24.2.6. Краткое описание библиотек

Система VTK является большим набором программных инструментов. В настоящее время система состоит из примерно 1,5 миллионов строк кода (включая комментарии, но не включая автоматически создаваемые программные обертки), а также из приблизительно 1000 классов на языке C++. Чтобы управлять такой сложной системой и уменьшить время, затрачиваемое на сбоку и компоновку, система была разделена на десятки подкаталогов. В таблице 24.1 перечислены эти подкаталоги и кратко описано, какие возможности представлены в каждой библиотеке.

Таблица 24.1: Подкаталоги системы VTK

Common	базовые классы системы VTK
Filtering	классы, используемые для управления потоком данных, идущих через конвейер
Rendering	рендеринг, выбор свойств, просмотр изображений и взаимодействие с изображением
VolumeRendering	технологии объемного рендеринга
Graphics	обработка трехмерной геометрии

GenericFiltering	обработка нелинейной трехмерной геометрии
Imaging	конвейер изображений
Hybrid	классы, требуемые для реализации графики и функциональных возможностей работы с изображениями
Widgets	сложные варианты взаимодействий
IO	вход и выход системы VTK
Infovis	визуализация информации
Parallel	параллельная обработка (контроллеры и коммуникаторы)
Wrapping	поддержка обверток для языков Tcl, Python и Java
Examples	обширные и хорошо документированные примеры

24.3. Оглядываясь назад / заглядывая вперед

Система VTK является чрезвычайно успешной системой. Хотя первая строка кода была создана в 1993 году, на момент написания этой статьи система VTK все еще продолжает сильно расти и темпы развития увеличиваются [2]. В этом разделе мы поговорим о некоторых усвоенных уроках и будущих исследованиях.

24.3.1. Управление ростом системы

Один из самых удивительных аспектов в приключении, связанном с системой VTK, является ее долголетие. Темпы развития обусловлены несколькими основными причинами:

- Продолжают добавляться новые алгоритмы и возможности. Например, недавним существенным добавлением стала информационная подсистема (Titan, первоначально разработанная лабораторией Sandia National Labs и Kitware). Также были добавлены дополнительные классы для рисования графиков и рендеринга, а также возможности для новых типов данных, используемых в научных исследованиях. Другим важным дополнением были трехмерные виджеты, используемые для взаимодействия. Наконец, к появлению новых возможностей в системе VTK ведет продолжающаяся эволюция рендеринга и обработки данных с использованием графических процессоров.
- Растущее влияние и использование системы VTK являются самоподдерживающимся процессом, в результате чего к сообществу добавляется еще больше пользователей и разработчиков. Например, система ParaView, которая является самым популярным приложением визуализации научных данных, построена на основе системы VTK и высоко ценится в сообществе высокопроизводительных вычислений. Система 3D Slicer является одной из основных биомедицинских вычислительных платформ, которая во многом построена на основе системы VTK и получает каждый год миллионы долларов финансирования.
- Разработка системы VTK продолжается. В последние годы в среду сборки системы VTK были интегрированы такие инструментальные средства, как CMake, CDash, CTest и CPack. Совсем недавно, репозитарий кода VTK был перемещен в Git и работа с ним усложнилась. Эти улучшения обеспечили, что система VTK держится в сообществе научных вычислений — на переднем крае разработки программного обеспечения.

Рост впечатляющий, что подтверждено самим созданием системы и служит хорошим предзнаменованием будущего VTK, но все это он чрезвычайно трудно поддается управлению. В результате в ближайшем будущем фокус в системе VTK будет больше сосредоточен на управлении ростом сообщества и развитием самой системы. Есть некоторые шаги в этом направлении.

Во-первых, созданы формальные структуры управления. Совет Architecture Review Board был создан для того, чтобы направлять развитие сообщества и технологий, уделяя особое внимание стра-

тегическим вопросам высокого уровня. Сообщество VTK также создало признанную команду лидеров Topic Leads, которая управляет развитием отдельных подсистем VTK.

Далее, есть планы на еще большую модуляризацию инструментальных средств частично в ответ на появившиеся возможности организации рабочего процесса, введенные с помощью git, но и признающие, что пользователи и разработчики обычно хотят работать с небольшими подсистемами инструментальных средств и не хотят собирать и компоновать весь пакет. Кроме того, для поддержки растущего сообщества, важно, чтобы поддерживался и тот вклад, который дают новые функциональные возможности и подсистемы даже в случае, если они не обязательно являются частью основного инструментального набора. Создавая слабо связанный набор модулей, можно добавить большое количество неосновных возможностей и сохранить стабильность базовой части.

24.3.2. Технологические дополнения

Кроме самого процесса разработки программного обеспечения, есть множество технологических инноваций, находящихся на стадии разработки

- Совместная обработка является такой возможностью визуализации, когда движок визуализации интегрируется в код, осуществляющий моделирование, и периодически создает отдельные поднаборы данных, используемых для визуализации. Эта технология значительно снижает необходимость вывода большого количества данных, представляющих собой полное решение.
- По-прежнему слишком сложным в системе VTK остается конвейер обработки данных VTK. Есть методы, направленные на упрощение и рефакторинг этой подсистемы.
- Среди пользователей становится все более популярной возможность напрямую взаимодействовать с данными. Хотя в системе VTK есть большой набор виджетов, появляется много других методов взаимодействия, в том числе методы, основанные на использовании сенсорных экранов, а также методы трехмерного взаимодействия. Механизмы взаимодействия продолжат свое достаточно быстрое развитие.
- Для разработчиков материалов и инженеров материаловедения все более значимой становится вычислительная химия. В систему VTK добавляется возможность осуществлять визуализацию и взаимодействовать с химическими данными.
- Система рендеринга, имеющаяся в VTK, была подвергнута критике за то, что она слишком сложна и из-за этого трудно создавать производные классы или поддерживать новые технологии рендеринга. Кроме того, в системе VTK прямо не поддерживается понятие сценического графа, которое опять же является тем, что просят многие пользователи.
- Наконец, постоянно появляются новые формы представления данных. Например, в области медицины - иерархические наборы волюметрических данных с различным разрешением (например, в конфокальной микроскопии с локальным увеличением).

24.3.3. Наука с открытыми возможностями

Наконец, Kitware и в целом все сообщество VTK являются приверженцами науки с открытыми возможностями. С прагматической точки зрения это способ сказать, что мы будем обнародовать открытые данные, открытые публикации и открытый исходный код — то, что нужно для создания репродуцируемых научных системы. Хотя система VTK уже давно распространяется с открытым исходным кодом и является открытой системой обработки данных, в ней не хватало документации. Хотя есть достойные книги [Kit10, SML06], также должны быть различные специальные способы сбора технических публикаций, в том числе и новых добавлений исходного кода. Мы улучшаем ситуацию путем разработки новых механизмов публикации, например, создав журнал *VTK Journal* [3], в котором представлены статьи, содержащие документацию, исходный код, данные и актуальные тестовые изображения. Журнал также позволяет получать автоматизированные оценки кода (с использованием процесса тестирования качества программ, имеющегося в системе VTK), а также публиковать обзоры подислок, создаваемые людьми.

24.3.4. Усвоенные уроки

Хотя система VTK является успешной, есть много того, что мы делали неправильно:

- *Модульность проекта:* Мы проделали хорошую работу для обеспечения модульности наших классов. Например, мы не делали чего-нибудь настолько глупого, как построение объекта размером в один пиксель, а создали более высокоуровневый класс `vtkImageClass`, под капотом которого обрабатываются массивы пиксельных данных. Однако в некоторых случаях мы сделали наши классы слишком высокоуровневыми и слишком сложными, поэтому во многих случаях нам приходится разделять их на мелкие части, и мы продолжаем этот процесс. Ярким примером является конвейер обработки данных. Вначале конвейер был реализован явно как взаимодействие объектов данных и объектов алгоритма. В конце концов мы поняли, что нам нужно было создать явный объект, осуществляющий выполнение работы конвейера, который будет координировать взаимодействие между данными и алгоритмами и реализовывать различные стратегии обработки данных.
- *Пропущенные ключевые концепции:* Одним из наших самых больших упущений явились то, что мы не стали широко использовать итераторы языка C++. Во многих случаях обход данных в системе VTK похож на обработку научных данных на языке программирования Fortran. Дополнительная гибкость, предоставляемая итераторами, позволила бы в системе добиться дополнительных преимуществ. Например, они очень удобны при обработке локальной области данных или только данных, удовлетворяющих некоторому критерию итерации.
- *Вопросы проектирования:* Конечно, есть длинный список проектных решений, которые не являются оптимальными. Мы боролись с конвейером выполнения данных, пережив несколько его поколений и каждый раз делая проект лучше. Также система рендеринга является слишком сложной для того, чтобы ее можно было развивать. Еще одна проблема возникла из первоначальной концепции системы VTK: мы видели ее лишь как систему визуализации, позволяющую только читать просматриваемые данные. Тем не менее, клиенты, пользующиеся ей в настоящий момент, часто хотят, чтобы была возможность редактирования данных, для чего необходимы совершенно другие структуры данных.

Одно из самых важных, касающееся системы с открытым исходным кодом, такой как VTK, является то, что многие из этих ошибок могут быть устраниены и будут со временем устраниены. У нас есть активное способное развиваться сообщество, которое ежедневно совершенствует систему, и мы ожидаем, что это будет продолжаться и в обозримом будущем.

Примечания

1. http://en.wikipedia.org/wiki/Opaque_pointer.
2. Смотрите анализ последнего варианта кода системы VTK по ссылке <http://www.ohloh.net/p/vtk/analyses/latest>.
3. <http://www.midasjournal.org/?journal=35>

25. Битва за Веснот

Программирование, как правило, рассматривается как простая деятельность по решению проблемы; у разработчика есть требования и он кодирует решение. О красоте часто судят по элегантности технической реализации или по ее эффективности; данная книга изобилует прекрасными примерами. Но код, кроме собственно вычислительных функций, может иметь огромное влияние на жизнь людей. Он может вдохновить людей принять участие в проекте и создать новое содержание. К сожалению, существуют серьезные барьеры, которые мешают людям участвовать в проекте.

Для того, чтобы пользоваться большинством языков программирования, необходимы существенные технические навыки, что для многих является недоступным. Кроме того, повышение доступ-

ности кода является для многих программ технически сложным и ненужным. Это редко ведет к аккуратному кодированию скриптов или к хорошо продуманным программным решениям. Обеспечение доступа к коду требует значительной предусмотрительности при разработке проекта и программы, из-за чего часто требуется следовать стандартам, которые континтуитивны нормальному программированию. Более того, в большинстве проектов полагаются на устоявшийся штат квалифицированных профессионалов, которые, как ожидается, работают на достаточно высоком уровне. Им не требуются дополнительные программистские ресурсы. Поэтому доступность кода, если она вообще рассматривается, становится второстепенной.

В нашем проекте, «Битва за Веснот» («Battle for Wesnoth»), с самого начала делалась попытка решить эту проблему. Данная программа является стратегической игрой в стиле фэнтези, созданной на основе модели с открытым исходным кодом и лицензией GPL2. Это был довольно внушительный успех с более, чем четырьмя миллионами скачиваний на момент написания статьи. Несмотря на такой впечатляющий показатель, мы считаем, что по-настоящему красивой гранью нашего проекта является модель разработки, позволяющая взаимодействовать и создавать свои решения группе добровольцев, обладающих крайне различными уровнями программистских навыков.

Повышение доступности кода не было всего лишь смутной целью, установленной разработчиками, а рассматривалось как условие, необходимое для выживания проекта. Подход с использованием открытого исходного кода в Wesnoth означал, что в проекте не следовало ожидать участия большого количества высококвалифицированных разработчиков. Создавая проект доступным широкому кругу участников, обладающих различной степенью квалификации, мы могли бы обеспечить ему длительную жизнестойкость.

С самых ранних итераций работы на проектом, наши разработчики пытались заложить основы для расширения доступности кода. Бессспорно, что это отразилось во всей архитектуре программы. Основные решения принимались с оглядкой, в значительной степени, на эту цель. В настоящей главе будет предоставлено подробное исследование нашей программы с особым вниманием к усилиям по увеличению доступности.

В первой части данной главы предлагается общий обзор приемов программирования, используемых в проекте, в том числе его языка, зависимостях и архитектуры. Во второй части мы сконцентрируемся на его уникальном языке описания данных, известном как Wesnoth Markup Language (WML). В ней объясняются конкретные функции WML с акцентом их влияния на игровые элементы. В следующей части будут рассматриваться вопросы многопользовательской реализации и использования внешних программ. В части, которая завершает настоящую главу будут приведены некоторыми заключительными наблюдениями, касающиеся нашей структуры и вопросов, связанных с расширением числа участников проекта.

25.1. Обзор проекта

На момент публикации настоящей статьи базовый движок Wesnoth, написанный на языке C++, состоял примерно из 200 000 строк кода. Сюда относится игровой движок, представляющий собой примерно половину этого кода, не содержащего какого-либо контента. В программе также можно задавать игровой контент при помощи уникального языка описания данных, известного как Wesnoth Markup Language (WML). Игра поставляется вместе с еще 250 000 строками кода на языке WML. На протяжении времени существования проекта это соотношение менялось. По мере того, как программа совершенствовалась, игровой контент, который был жестко запрограммирован на C++, всё больше и больше переписывался таким образом, чтобы для определения действий программы можно было использовать язык WML. На рис.25.1 приведен общий вид архитектуры программы; областями серого цвета выделены те части, которые поддерживаются разработчиками проекта Wesnoth, а те, которые выделены белым, являются внешними зависимостями.

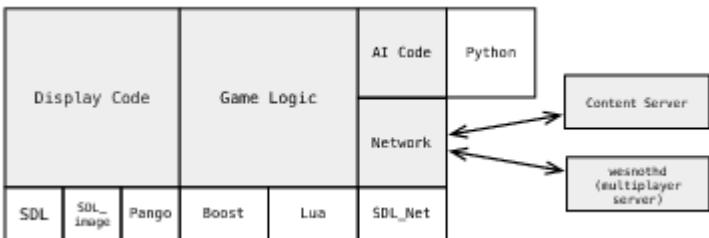


Рис.25.1: Архитектура программы

В целом, в большинстве случаев проект старается минимизировать количество зависимостей с тем, чтобы увеличить переносимость приложения. Преимуществом этого является уменьшение сложности программы, а также то, что разработчикам не требуется изучать нюансы большого количества интерфейсов API, разработанных третьими сторонами. В то же время, осторожное использование некоторых зависимостей может, на самом деле, помочь добиться такого же эффекта. Например, в Wesnoth используется слой Simple Directmedia Layer (SDL) для работы с видео, с вводом/выводом и для обработки событий. Слой SDL был выбран за то, что он прост в использовании, и в нем предоставлен обычный интерфейс ввода/вывода, реализованный для многих платформ. Это позволяет обеспечивать переносимость множества платформ, а не кодировать на различных платформах альтернативные варианты для конкретных интерфейсов API. За это приходится платить, т.к. это оказывается сложнее, чем пользоваться преимуществами некоторых специальных возможностей, имеющихся в ряде платформ. В SDL также есть семейство дополнительных библиотек, которые в проекте Wesnoth используются для различных целей:

- **SDL_Mixer** — для микширования аудиосигналов и звука
- **SDL_Image** - для загрузки изображений в формате PNG и в других форматах
- **SDL_Net** - для работы с сетью

Кроме этого, в Wesnoth используется несколько других библиотек:

- **Boost** - для различных передовых возможностей языка C++
- **Pango** и **Cairo** - для использования шрифтов различных языков
- **zlib** - для сжатия данных
- **Python** и **Lua** - для поддержки работы со скриптами
- **GNU gettext** - для использования разнообразных языков

Повсюду в движке Wesnoth используются объекты WML, т.е. практически повсеместны строковые словари с дочерними элементами. Множество объектов может создаваться из элементов WML, а также может быть сериализовано в элементы WML. В некоторых частях движка данные хранятся в таком формате, который базируется на словарях WML, причем они непосредственно интерпретируются, а не преобразовываются в структуру данных языка C++.

В Wesnoth используется несколько важных подсистем, большинство из которых сделаны настолько автономными, насколько это было возможным. Такая сегментированная структура имеет преимущества для обеспечения доступности. Любой. Кому это будет интересно, может достаточно работать над кодом из конкретной части программы и вносить туда изменения, не повреждая остальную часть программы. К числу основных подсистем относятся следующие:

- Синтаксический анализатор языка WML с препроцессором.
- Базовые модули ввода/вывода, абстрагирующие библиотеки и системные вызовы, лежащие ниже — модуль работы с видео, модуль работы со звуком и модуль работы с сетью.
- Модуль графического пользовательского интерфейса, содержащий реализации виджетов кнопок, списков, меню и т.д.
- Модуль отображения — для прорисовывать игрового поля, юнитов, анимации и т.д.
- Модуль искусственного интеллекта.

- Модуль поиска путей, в состав которого входит много вспомогательных функций, используемых для работы с игровым полем, состоящим из шестиугольников.
- Модуль генерации карт - для генерации произвольных вариантов карт различного вида.

Также существуют модули для управления различными этапами игрового процесса:

- Модуль титульного экрана - для показа титульного экрана.
- Модуль сюжетной линии - для показа роликов-вставок.
- Модуль главного меню игры - для показа и подключения к играм многопользовательского сервера.
- Модуль «play game», который контролирует основной ход игры.

Модуль «play game» и основной модуль отображения являются самыми большими в Wesnoth. Их назначение определено наименее четко, поскольку функции меняются постоянно и для них сложно составить четкую спецификацию. Поэтому в течение всего времени существования программы для этих модулей часто возникал риск их реализации в виде антипаттерна Blob, т. е. превращения их в громадные доминирующие сегменты без строго определенного поведения. Код в этих модулях регулярно просматривается с целью определить, можно ли какую-нибудь из его частей выделить в отдельный модуль.

Есть также вспомогательные функции, которые являются частью общего проекта, но реализованы отдельно от основной программы. К ним относится многопользовательский сервер, который позволяет подключаться к многопользовательским сетевым играм, а также сервер контента, который позволяет пользователям загружать их контент на общий сервер и делать его доступным для других. Оба они написаны на языке C++.

25.2. Wesnoth Markup Language — язык разметки Wesnoth

Поскольку Wesnoth является расширяемым игровым движком, в нем используется простой язык описания данных, позволяющий сохранять и загружать все игровые данные. Хотя сначала рассматривалось использование языка XML, мы решили, что нам нужно что-нибудь более дружественное для пользователей, которые технически подготовлены слабее, и менее строгое для описания визуальных данных. Поэтому мы разработали свой собственный язык описания данных, называемый Wesnoth Markup Language (WML). Он был разработан в расчете на пользователей, имеющих самую слабую техническую подготовку: мы надеялись, что даже те пользователи, для которых языки Python и HTML кажутся страшными, смогут разобраться в файле WML. Все игровые данные Wesnoth хранятся на языке WML, в том числе определения юнитов, описания кампаний, сценарии, определения пользовательского интерфейса и другие настройки игровой логики.

В языке WML используются такие же самые базовые элементы, что и в языке XML: элементы и атрибуты, хотя в нем не поддерживается использование текста внутри элементов. Атрибуты WML представлены просто как словарь, осуществляющий отображение строк в строки, а за интерпретацию атрибутов отвечает логика программы. Ниже показан простой пример на языке WML, с помощью которого в игре приводится сокращенное определение юнита Elvish Fighter (Эльфийский Воин):

```
[unit_type]
  id=Elvish Fighter
  name=_ "Elvish Fighter"
  race=elf
  image="units/elves-wood/fighter.png"
  profile="portraits/elves/fighter.png"
  hitpoints=33
  movement_type=woodland
  movement=5
  experience=40
```

```

level=1
alignment=neutral
advances_to=Elvish Captain,Elvish Hero
cost=14
usage=fighter
{LESS_NIMBLE_ELF}
[attack]
    name=sword
    description=_"sword"
    icon=attacks/sword-elven.png
    type=blade
    range=melee
    damage=5
    number=4
[/attack]
[/unit_type]

```

Поскольку в Wesnoth очень важна интернационализация, в языке WML есть ее непосредственная поддержка: значения атрибутов, начинающиеся со знака подчеркивания, являются переводимыми. Когда происходит синтаксический разбор языка WML, все переводимые строки с помощью GNU gettext преобразовываются в переведенные версии строк.

Вместо того, чтобы использовать много различных документов WML, в Wesnoth выбран подход, в котором все основные игровые данные передаются в игровой движок просто в виде единого документа. Это позволяет для работы с документом использовать только одну глобальную переменную, а когда в игре происходит загрузка, например, определения юнитов, выполнять поиск элементов с именем `unit_type`, находящихся внутри элемента `units`.

Не смотря на то, что все данные хранятся в виде одного концептуально единого документа WML, было бы громоздко хранить весь этот документ в одном файле. Поэтому в Wesnoth используется препроцессор, который перед тем, как начнется синтаксический разбор, проходит по всем файлам WML. Этот препроцессор позволяет в файл добавлять содержимое другого файла или целого каталога. Например:

```
{gui/default/window/}
```

будет добавлять содержимое всех файлов `.cfg`, находящихся в каталоге `gui/default/window/`.

Поскольку описание в WML может быть очень подробным, в препроцессоре также можно использовать макросы, которые нужны для более компактных определений. Например, обращение к `LESS_NIMBLE_ELF` в определении Эльфийского Воина (Elvish Fighter) является вызовом макроса, который делает некоторых эльфийских юнитов менее ловкими при определенных условиях, например, когда они размещены в лесу:

```

#define LESS_NIMBLE_ELF
[defense]
    forest=40
[/defense]
#endif

```

Данная конструкция обладает тем преимуществом, что движок не считает нужным разбираться с тем, как документ WML разбит на отдельные файлы. Это ответственность лежит на авторах документа WML, которые решают, как структурировать документ и как разделить его на отдельные файлы и каталоги.

Когда игровой движок загружает документ WML, он, в соответствии с различными настройками игры, также определяет некоторые значения настроек для препроцессора. Например, для кампании в Wesnoth можно определять различные уровни сложности, причем различные настройки каждого из них приведут к тому, что для препроцессора будут определены различные значения настроек.

Например, обычным способом изменения сложности игры является изменение количества ресурсов, предоставляемых оппоненту (определенным количеством золота). Чтобы это упростить, используется макрос, который определен следующим образом:

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
#define EASY
gold={EASY_AMOUNT}
#endif
#define NORMAL
gold={NORMAL_AMOUNT}
#endif
#define HARD
gold={HARD_AMOUNT}
#endif
#endif
```

Этот макрос может быть вызван, например, внутри определения оппонента как {GOLD 50 100 200} с тем, чтобы указать, сколько в зависимости от уровня сложности он получит золота.

Поскольку XML обрабатывается в зависимости от настроек, то если какое-либо значение, прилагаемое к документу WML, изменится во время его исполнения движком Wesnoth, весь документ WML должен быть повторно загружен и обработан. Например, когда пользователь запускает игру, то загружается документ WML и среди всего прочего загружается и список доступных кампаний. Но, если затем пользователь выбирает кампанию и выбирает определенный уровень сложности, например, легкий уровень, то весь документ будет перезагружен с установленным параметром EASY.

Такая конструкция удобна тем, что в одном документе содержатся все игровые данные и что с помощью значений настроек можно легко сконфигурировать документ WML. Но в Wesnoth, как у успешного проекта, появляется всё больше и больше контента, в том числе контента, доступного для загрузки, который, в конце концов, весь оказывается в дереве основного документа, что означает, что размер окончательно получившегося документа WML будет составлять много мегабайтов. Из-за этого возникают проблемы с производительностью Wesnoth: на некоторых компьютерах загрузка документа может доходить до минуты, приводя в игре к задержкам каждый раз, когда документ необходимо перезагрузить. Кроме того, при этом используется значительный объем памяти. Против этого предпринимаются некоторые меры: когда кампания загружается, то для этой кампании есть уникальное значение, определяемое в препроцессоре. Это означает, что когда нужна эта кампания, то будет использован весь тот контент, принадлежность которого к кампании определяется с помощью #ifdef.

Кроме того, в Wesnoth используется система кеширования, которая кеширует препроцессорную версию документа WML для заданного набора ключевых определений. Естественно, что такая кеширующая система должна проверять даты изменения файлов WML для того, чтобы в случае, если какой-нибудь из них будет изменен, перегенерировать закешированный документ.

25.3. Юниты в Wesnoth

Главными героями в Wesnoth являются его юниты. Эльфийский Воин (Elvish Fighter) и Эльфийский Шаман (Elvish Shaman) могут сражаться против Воина-Тролля (Troll Warrior) и Орка-Пехотинца (Orcish Grunt). У всех юнитов одинаковое базовое поведение, но многие из них имеют специальные способности, которые изменяют нормальный ход игры. Например, тролль с каждым ходом может регенерировать часть своего здоровья, Эльфийский Шаман может замедлять действия своих противников с помощью запутывающих корней, а Лешего (Wose) нельзя увидеть, когда тот находится в лесу.

Как лучше всего представить это в движке? Заманчиво создать базовый класс `uni` на языке C++, от которого наследовать различные виды юнитов. Например, класс `wose_unit` можно было бы наследовать из `unit`, и `unit` мог бы иметь виртуальную функцию `bool is_invisible() const`, которая возвращает значение `\code{false}`, и которую `wose_unit` переопределяет с тем, чтобы она возвращала значение `\code{true}` в случае, если данный юнит сейчас находится в лесу.

Такой подход работал бы довольно хорошо в игре с ограниченным набором правил. К сожалению, Wesnoth является довольно большой игрой, а такой подход нельзя достаточно просто расширить. Если кто-нибудь в случае использования такого подхода захочет добавить новый вид юнитов, то для этого потребуется добавление в игру нового класса C++. Кроме того, такой подход не позволяет достаточно хорошо комбинировать различные характеристики: что если бы у вас был юнит, который умеет регенерировать, замедлять врагов при помощи сетки, и который невидим в лесу? Вам бы пришлось написать совершенно новый класс, в котором бы дублировался код других классов.

Система юнитов, имеющаяся в Wesnoth, для того, чтобы справиться с этой задачей, совсем не использует наследование. Вместо этого используется класс `unit`, в котором представлены экземпляры юнитов, а также класс `unit_type`, с помощью которого представлены неизменяемые характеристики, которыми обладают все юниты определенного типа. Класс `unit` содержит ссылку на тип объекта, которым он является. Все возможные объекты класса `unit_type` хранятся в глобальном словаре, загружаемом при загрузке главного документа WML.

Тип юнита содержит список всех способностей данного юнита. Например, у Тролля есть способность «регенерация», которая позволяет ему с каждым ходом восстанавливать жизни. У Саурийского стрелка (Saurian Skirmisher) есть способность «стрельба», которая позволяет ему пробиваться сквозь цепи врагов. Распознавание этих способностей встроено в движок, например, алгоритмы поиска пути будут проверять, выставлен ли у юнита флаг «стрельба» с тем, чтобы знать может ли юнит пробиваться сквозь цепи врагов. Такой подход позволяет с помощью только редактирования текста на WML добавлять новые юниты, у которых может быть любая комбинация способностей, определенных в движке. Конечно, при этом не допускается без модификации движка добавлять совершенно новые способности и поведение юнитов.

Кроме того, каждый юнит в Wesnoth может обладать любым количеством способов атак. Например, Эльфийский Лучник (Elvish Archer) может, стреляя из лука, атаковать на дальние расстояния, а вблизи атаковать используя меч. Каждая из атак наносит различный урон различного вида и различной величины. Для представления типов атак существует класс `attack_type` и У каждого экземпляра класса `unit_type` есть список возможных типов атак в виде списка типов `attack_type`.

Чтобы обеспечить каждому юниту больше индивидуальности, в Wesnoth есть такое понятие, как «особенности». В момент рекрутинга большинству юнитов назначаются две особенности, выбираемые из предопределенного списка случайным образом. Например, сильные юниты наносят больше урона при ближней атаке, в то время как умным юнитам потребуется набирать меньше опыта для того, чтобы перейти на следующий уровень. Также юниты могут по ходу игры получать «снаряжение», которое может сделать их более сильными. Например, это может быть меч, поднятый юнитом, с которым его атаки будут наносить больший урон. Чтобы реализовать «особенности» и «снаряжение», в Wesnoth разрешается определять модификации для юнитов, которые являются определенными с помощью WML вариациями статистических свойств юнита. Например, особенность-сила позволяет сильным юнитам наносить больший урон вблизи, но не в случаях дальних атак.

Разрешить полностью определять поведение юнитов с помощью языка WML могло бы быть замечательной целью, поэтому было бы правильным объяснить, почему в Wesnoth эта цель никогда не была достигнута. Если бы было разрешено произвольное поведение юнитов, то язык WML должен был бы быть намного более гибкими, чем он есть сейчас. Вместо того, чтобы использовать WML

как язык, предназначенный для описания данных, его бы пришлось так расширить, что он бы стал полноценным языком программирования, что отпугнуло бы многих участников проекта.

Кроме того, в игре участвует модуль искусственного интеллекта Wesnoth AI, написанный на языке C++, который распознаёт различные способности юнитов. Он учитывает регенерацию, невидимость, и т.д., и старается управлять своими юнитами так, чтобы получать наибольшую выгоду от этих способностей. Даже если бы новые способности юнитов можно было добавлять с помощью языка WML, было бы очень сложно сделать модуль искусственного интеллекта настолько умным, чтобы распознавать и использовать такие способности. Реализация способностей была бы неполной, если бы модуль искусственного интеллекта не мог бы их использовать. Также было бы странным, если бы после определения новой способности на языке WML приходилось бы изменять код модуля искусственного интеллекта, написанный на языке C++. Поэтому наличие юнитов, определяемых на языке WML, но обладающих способностями, жестко запрограммированными в движке, является разумным компромиссом, который в специфических условиях проекта Wesnoth работает лучше всего.

25.4. Реализация в Wesnoth многопользовательских игр

В многопользовательском варианте игры в Wesnoth используется максимально простая реализация. В ней делаются попытки уменьшить вероятность вредоносных атак на сервер, но не слишком много делается с целью предотвращать мошенничество в игре. Каждый ход в Wesnoth, будь то перемещение юнита, атака врага, рекрутинг нового юнита и т.д., может быть представлен в виде элемента языка WML. Например, команда перемещения юнита может быть представлена в виде WML следующим образом:

```
[move]
  x="11,11,10,9,8,7"
  y="6,7,7,8,8,9"
[/move]
```

Здесь показан путь, по которому движется юнит в результате выполнения команд, отдаваемых игроком. Далее, у игры есть возможность выполнять любые команды WML, которые она получает. Это очень удобно, поскольку если сохранить начальное состояние игры, а также все последующие команды, то может быть сохранен весь ход игры. Возможность повторять весь ход игр полезна как для самих игроков, чтобы можно было наблюдать за тем, как играет кто-то другой, так и для того, чтобы делать сообщения об ошибках определенного вида.

При реализации в Wesnoth многопользовательских игр, мы решили, что сообщество будет пытаться сосредоточиться на дружественных, казуальных играх. Вместо того, чтобы вступать в техническую борьбу с антиобщественными кракерами, пытающимися обмануть систему профилактики взлома, проект просто не будет стараться предотвращать мошенничество. Анализ других многопользовательских игр показал, что конкурентоспособные системы рейтинга были ключевым источником антиобщественного поведения. Умышленное отсутствие таких функций на сервере значительно снижает мотивацию у тех, кто пытается обмануть. Кроме того, модераторы стараются поощрять более позитивные отношения в игровом сообществе, когда игроки находят личное взаимопонимание с другими игроками, а затем играют с ними. В результате акцент перемещается с конкуренции на взаимоотношения. Результат данных попыток был признан успешным, т.к. количество попыток вредоносного взлома игры значительно сократилось.

Реализация многопользовательской игры в Wesnoth представляет собой типичную клиент-серверную инфраструктуру. Сервер, также известный как wesnothd, принимает соединение от клиента Wesnoth и отсылает клиенту краткое описание доступных игр. Wesnoth покажет игроку экран главного меню, где игрок сможет присоединиться к существующей игре или создать новую игру, к которой присоединятся другие. Когда все игроки присоединились к игре и игра началась, каждый клиент Wesnoth будет генерировать команды WML, описывающие действия, которые совершает

игрок. Эти команды каждый клиент отсылает на сервер, а затем сервер передает их другим клиентам, участвующим в игре. Таким образом, сервер выступает в роли очень простого и легковесного ретранслятора. Затем на других клиентах используется система проигрывания команд для того, чтобы выполнять полученные команды WML. Поскольку Wesnoth является пошаговой игрой, для всех сетевых коммуникаций используется протокол TCP/IP.

Данная система также легко позволяет наблюдателям следить за игрой. Наблюдатель может присоединиться к игре после того, как она началась. В этом случае сервер передаст наблюдателю WML, описывающий начальное состояние игры, а затем историю всех команд, которые были выполнены после начала игры. Это позволяет новым наблюдателям увидеть, что происходило в игре ранее. Они могут видеть всю историю игры, однако требуется некоторое время, чтобы наблюдатель мог догнать текущее состояние игры. Историю команд можно прокручивать в ускоренном режиме, но всё равно это занимает некоторое время. Альтернативой могло бы быть, если бы один из клиентов генерировал снимок текущего состояния игры в виде WML и отсыпал бы новому наблюдателю, однако такой подход обременял бы клиентов накладными расходами, связанными с наличием наблюдателей, и мог бы способствовать возникновению состояний типа DoS-атак в случае, когда к игре присоединялось бы большое количество наблюдателей.

Конечно, поскольку клиенты Wesnoth не пользуются совместно каким-либо игровым статусом, а только отсылают команды, очень важно чтобы все они играли по одним правилам. Сервер разделяет клиентов по версиям, и друг с другом могут играть только игроки с одинаковыми версиями игры. Игроки сразу же получают уведомление в случае, если их версия игры устарела в сравнении с другими. Это также является полезным способом предотвращения мошенничества. Не смотря на то, что клиенту легко смошенничать, изменив своего клиента, о любом различии в версиях сразу будет сообщено игрокам, участвующим в игре.

25.5. Заключение

Нам кажется, что красота Battle for Wesnoth, как программы, состоит в том, насколько участие в ее развитии сделано доступным для широкого круга участников. Чтобы достичь этой цели, проект часто шел на компромиссы, из-за которых код выглядит совсем не элегантно. Важно заметить, что многие из более талантливых программистов проекта относятся к языку WML с неодобрением, из-за неэффективности его синтаксиса. Тем не менее, этот компромисс лежит в основе одного из самых больших достижений проекта. Сейчас Wesnoth может похвастаться сотнями игр с различными кампаниями и разнообразными эпохами, написанными пользователями, у которых до этого не было или почти не было опыта в программировании. Более того, это вдохновило многих, использовавших данный проект как средство обучения, выбрать своей профессией программирование. Всё это является вполне осозаемыми достижениями, с которыми могут сравняться лишь некоторые программы.

Одним из ключевых уроков, которые читателю стоит вынести из истории с Wesnoth, это учет тех проблем, с которыми сталкиваются менее опытные программисты. Необходимо осознавать, что именно препятствует участникам проекта действительно писать код и развивать свои навыки. Например, кто-то хотел бы помочь развиваться программе, но он не обладает какими-либо навыками в программировании. Специальные технологические текстовые редакторы, такие как `emacs` или `vim`, обладают значительной кривой обучения, которая может показаться для такого участника пугающей. Поэтому был разработан язык WML, файлы на котором можно открывать в обычном текстовом редакторе, что дает всем желающим возможность принять участие в проекте.

Однако, увеличение доступности кода не является легко достижимой целью. Нет простых и быстрых правил для улучшения доступности кода. Скорее всего для этого требуется баланс между различными факторами, которые могут иметь негативные последствия, о чем должно быть осведомлено сообщество проекта. Это становится очевидным, если посмотреть на то, как наша программа

справлялась с зависимостями. В одних случаях зависимости могут повысить порог вхождения, а в других - они могут облегчить участие в проекте. Каждый такой случай нужно рассматривать отдельно.

Также нужно быть аккуратным с тем, чтобы не переоценить некоторые из успехов проекта Wesnoth. В проекте можно было воспользоваться некоторыми преимуществами, которые, скорее всего, не удастся воспроизвести в других программах. Доступность кода для более широкого круга участников является частично результатом того, что в программе есть возможности ее настройки. В этом плане в Wesnoth были некоторые преимущества, как у проекта с открытым исходным кодом. Юридически лицензия GNU позволяет любому открыть существующий файл, разобраться в том, как он работает, и внести изменения. В этой культуре поощряются экспериментирование, изучение и совместное использование знаний, что может оказаться неприемлемым для других программ. Не смотря ни на что, мы надеемся, что определенные элементы все-таки пригодятся всем разработчикам и помогут им в их попытках отыскать красоту в программировании.