Управление проектами с помощью GNU make

Роберт Мекленбург

7 апреля 2020 г.

Введение

Путь к третьему изданию

Моё первое знакомство с *make* произошло, когда я был студентом в университете Беркли в 1979 году. Мне посчастливилось работать с «новейшим» оборудованием: компьютером DEC PDP 11/70, имевшим 128 килобайт оперативной памяти и терминал ADM 3a «glass tty», управляемым операционной системой Berkeley Unix, обслуживавшей помимо меня ещё 20 пользователей! Вспоминается, как много времени отнимал вход в систему - пять минут с момента ввода имени пользователя до момента получения приглашения командного интерпретатора.

После окончания университета я снова стал пользователем Unix лишь в 1984 году. На этот раз я был программистом в Исследовательском Центре NASA им. Д.С. Эймса. Мы с моими коллегами выпустили одну из первых систем Unix на базе микрокомпьютеров 68000, которые имели мегабайт оперативной памяти, работали под управлением Unix Version 7 и поддерживали одновременную работу до шести пользователей. Моим последним проектом в NASA была интерактивная система анализа данных со спутника, написанная на языке C с использованием таких инструментов как yacc/lex и, конечно же, *make*.

В 1988 году я вернулся в университет и работал над проектом программы моделирования геометрических фигур с помощью сплайнов. Система состояла из примерно 120000 строк кода на языке C и размещалась в двадцати (или около того) исполняемых файлах. Система собиралась при помощи *makefile*-шаблонов, расширяемых вспомогательной программой *genmakefile* (близкой по духу программе *imake*) в обычные *makefile*'ы. *genmakefile* осуществляла включение файлов, условную компиляцию и специфическую логику управления исходным кодом и деревьями каталогов бинарных файлов. В те дни все были уверены: для того, чтобы *make* был полноценным инструментом сборки программного обеспечения, ему требуется подобная обёртка. Несколькими годами ранее я открыл для себя проект GNU и их версию *make*. Тогда меня посетила мысль, что программа-обёртка, возможно, не является необходимостью. Я переделал систему сборки так, чтобы она не использовала шаблоны и генератор. К моему огорчению, мне пришлось поддерживать систему на протяжении последующих четырёх лет (поведенческий шаблон,

по глупости повторяемый мной по сей день). Система сборки была портирована на пять различных Unix-подобных операционных систем и включала в себя раздельные деревья каталогов исходных и бинарных файлов, автоматизированные ночные сборки и возможность создания разработчиками частичных рабочих копий (partial checkouts) с автоматическим заполнением недостающих объектов.

Моя следующая интересная встреча с *make* произошла в 1996 году при работе над коммерческой CAD-системой. Моей задачей был перенос двух миллионов строк кода на языке C++ (и ещё примерно 400 000 строк кода на языке Lisp) с Unix на Windows NT с использованием компилятора C++ корпорации Microsoft. Именно тогда я открыл для себя проект Cygwin. В качестве важного субпродукта переноса я переработал свою систему сборки так, чтобы она работала на операционной системе Windows NT. Новая система сборки также поддерживала раздельные деревья каталогов исходных и бинарных файлов, множество версий Unix, несколько вариантов графических оболочек, автоматизированные ночные сборки и тесты, а также возможность создания частичных рабочих копий и использования справочной сборки.

В 2000 году я стал одним из разработчиком системы управления экспериментальными данными. Основным языком разработки был язык Java. Среда разработки в этом проекте очень сильно отличалась от тех, в которых я работал много лет. Большая часть программистов имела опыт программирования под Windows, и, похоже, язык Java был их первым языком программирования. Среда сборки практически полностью состояла из файла проекта, созданного коммерческой интегрированной средой разработки для Java. Не смотря на то, что файл проекта был тщательно проверен, среда сборки редко работала «из коробки», и программисты часто сидели в своих кубиках, исправляя проблемы со сборкой системы.

Естественно, я начал писать собственную систему сборки, основанную на *make*, но меня ожидало неприятное удивление. Оказалось, что большая часть разработчиков весьма неохотно пользовались командным интерпретатором. Более того, многие не владели такими концепциями, как переменные окружения и аргументы командной строки, или пониманием инструментов, используемых для сборки программного обеспечения. Интегрированные среды разработки прятали все эти детали от пользователей. Чтобы учесть все эти тонкости, я усложнил свою систему сборки, добавив развёрнутые описания ошибок, проверку предусловий, управление конфигурацией машины разработчика и поддержку интегрированных сред разработки.

В процессе реализации я прочитал руководство по GNU *make* несколько десятков раз. В поисках материала я наткнулся на второе издание этой книги. Оно было наполнено очень полезной информацией, но, к сожалению, опускало многие детали работы GNU *make*. И это не удивительно, учитывая возраст издания. Книга выдержала испытание временем, но на момент 2003 года она нуждалась в обновлении. Третье издание сфокусировано преимущественно на GNU *make*. Как писал

Пол Смит (разработчик, поддерживающий GNU *make*), «не тратьте силы, пытаясь написать переносимые *makefile*'ы, используйте переносимую версию *make*!».

Что нового в этом издании

Практически весь материал книги обновлён. Я разделил его на три части.

Часть I, *Основные концепции*, представляет достаточно детальный обзор функциональности GNU *make* и способов её использования.

Глава 1 — это краткое введение в *make*, содержащее простой, но законченный пример. Она объясняет базовые концепции *make*, такие как цели и реквизиты, и описывает синтаксис *makefile*'ов. Этого должно быть достаточно, чтобы вы научились составлять свои первые *makefile*'ы.

В главе 2 обсуждаются структура и синтаксис правил. Даётся детальное описание явных и шаблонных правил, в том числе старомодных суффиксных правил. Здесь же обсуждаются специальные цели и основы автоматического определения зависимостей.

Глава 3 содержит сведения о простых и рекурсивных переменных. Здесь также обсуждается обработка *makefile*'ов, события, приводящие к подстановке переменных и директивы для условной обработки *makefile*'а.

В главе 4 рассматривается множество встроенных функций GNU *make*. Здесь же вводятся определяемые пользователем функции и различные примеры их использования, начиная от тривиальных иллюстраций и заканчивая сложными концепциями.

Глава 5 объясняет детали применения сценариев сборки, покрывая такие аспекты, как их синтаксический разбор и выполнение. Здесь же обсуждаются модификаторы команд, проверка кодов возврата и окружение. Мы исследуем проблемы конечности длины командной строки и способы их преодоления. На этом этапе вы будете знать все возможности *make*, используемые в этой книге.

Часть II, *Специализированные вопросы*, покрывает такие большие темы, как использование *make* в больших проектах, переносимость и отладка.

В главе 6 обсуждаются многие проблемы, возникающие при сборке больших систем с помощью *make*. Первой темой будет использование рекурсивного вызова *make* и способы описания всего процесса сборки в единственном нерекурсивном *makefile*'е. Также будут рассмотрены другие проблемы, возникающие в больших системах, например, расположение файлов в файловой системе, управление компонентами, автоматическая сборка и тестирование.

В главе 7 обсуждаются вопросы переносимости *makefile*'ов, преимущественно между различными версиями Unix и Windows. В деталях рассматривается эмулятор среды Unix — проект Cygwin — и связанные проблемы, возникающие при использовании непереносимых инструментов и возможностей файловой системы.

Глава 8 содержит специфические примеры разделения деревьев каталогов исходных и объектных файлов, а также способ создания деревьев каталогов исходного кода с правами «только для чтения». Снова рассматривается анализ зависимостей, ударение делается на решения, зависимые от языка. Эта глава вместе со следующей неразрывно связана с вопросами, поднятыми в главе 6.

В главе 9 объясняется, как применить *make* к среде разработки приложений на языке Java. Описываются техники управления переменной `CLASSPATH`, компиляции большого числа исходных файлов, создания JAR-архивов и конструирования EJB.

Глава 10 начинается с обзора производительности некоторых операций *make*, дающего почву для рассуждений о производительности *makefile*'ов. Обсуждаются техники обнаружения и устранения критических участков кода, влияющих на производительность системы в целом. Детально описывается возможность параллельного выполнения команд при помощи GNU *make*.

В Главе 11 представлены два примера реальных *makefile*'ов. Первый из них использовался при создании этой книги. Он интересен по двум причинам. Во-первых, степень автоматизации действий в нём чрезвычайно высока. Во-вторых, он демонстрирует применение *make* в нетрадиционной для него области. В качестве второго примера используются выдержки из системы сборки ядра Linux 2.6 — *kbuild*.

Глава 12 погружает нас в искусство отладки *makefile*'ов. Здесь мы увидим техники определения действий, незримо выполняемых *make*, а также способы облегчения разработки.

Часть III, *Дополнения*, включает вспомогательный материал.

Приложение А предоставляет собой справочник по опциям командной строки утилиты GNU *make*

В Приложении Б указаны пределы использования GNU *make* по двум характеристикам: управлении структурами данных и вычислении производительности.

Приложение В содержит текст лицензии *GNU Free Documentation License*, под которой распространяется текст этой книги.

Типографские соглашения

В книге приняты следующие типографские соглашения:

Наклонный шрифт

означает новые термины, URL, адреса электронной почты, имена и расширения файлов, пути в файловой системе и имена каталогов.

`Моноширинный шрифт`

означает исходный код, команды интерпретатора, опции командной строки, содержимое файлов и вывод команд.

`Моноширинный полужирный шрифт`

означает команды или другой текст, который должен быть набран пользователем.

`Моноширинный наклонный шрифт`

означает текст шаблона, который должен быть заменён пользователем.

Использование примеров кода

Эта книга была создана, чтобы помогать вам выполнять вашу работу. В целом, вы можете использовать код, приведённый в этой книге, в ваших программах и документации. Вам не нужно связываться с издательством и просить разрешения, пока Вы не решите воспроизвести значительный кусок кода. Например, написание программы, использующей несколько отрывков кода из книги не требует разрешения. Для продажи или распространения электронных носителей, содержащих примеры из книги, *требуется* разрешение. Для ответа на вопросы других людей со ссылкой на эту книгу не требуется разрешение. Для внедрения значительного числа примеров кода из этой книги в документацию вашего продукта *требуется* разрешение.

Если вам кажется, что ваше использование примеров кода из это книги выходит за рамки законного использования, свяжитесь с издательством.

Благодарности

Я хотел бы поблагодарить Ричарда Столлмена за то, что он подарил мне мечту и веру в её исполнение. Естественно, без Пола Смита GNU *make* не существовал бы сейчас в том виде, в котором мы используем. Спасибо тебе, Пол.

Я хотел бы поблагодарить моего редактора, Энди Орама, за его неоценимую помощь и энтузиазм.

Отдельной благодарности заслуживает компания Cimarron Software, которая предоставила мне среду, воодушевившую меня начать этот проект. Хотелось бы также поблагодарить компанию Realm Systems, предоставившую мне среду, воодушевившую меня закончить проект. В частности, я хотел бы поблагодарить Дуга Адамсона, Кэти Андерсон и Питера Букмена.

Спасибо моим рецензентам, Симону Геррети, Джону Макдональду и Полу Смиту, предоставившим много глубоких замечаний и поправившим много досадных ошибок.

Следующие люди заслуживают благодарность за значительный вклад в эту работу: Стив Байер, Ричард Богарт, Бет Кобб, Джули Дэйли, Дэвид Джонсон, Эндрю Мортон, Ричард Паймнтел, Брайэн Стивенс и Линус Торвальдс. Большое

спасибо группе заговорщиков, обеспечивших мне стабильный и спокойный островок в бушующих штормами морях: Кристине Дилэйни, Тони Ди Сера, Джону Мажору и Даниелю Ридинг.

Наконец, выражаю глубокую благодарность и любовь своей жене, Мэгги Кэстен, и нашим детям, Вильяму и Джэймсу, за их поддержку, воодушевление и любовь на протяжении последних шести месяцев. Спасибо за то, что вы были со мной.

Оглавление

Часть I

Основные концепции

В этой части мы сфокусируемся на возможностях *make* и их правильном использовании. Мы начнём с краткого введения и обзора, которых должно быть достаточно для того, чтобы вы могли написать свой первый *makefile*. Главы этой части покрывают правила, переменные, функции и сценарии сборки.

После прочтения этой части у вас будет законченное представление о том, как работает GNU *make*.

Глава 1

Как написать простой *makefile*

Механизм создания программ обычно довольно прост: редактирование исходного кода, компиляция в исполняемый файл и отладка полученной программы. Несмотря на то, что компиляция считается рутинной операцией, будучи проведённой некорректно, она может породить ошибки, на исправление которых у программиста уйдёт очень много времени. Многие разработчики испытывают удивление, когда после исправления какой-нибудь функции и запуска нового варианта программы видят, что их изменения не исправляют ошибок. Позже они обнаружат, что их модифицированный код никогда не выполнялся по причине ошибки процедуры компиляции, компоновки или сборки JAR-архива. Более того, с ростом сложности программ эти рутинные задачи становятся источниками всё более сложных ошибок, поскольку одновременно могут разрабатываться различные версии приложения, например, для разных платформ или с использованием разных версий библиотек.

Программа *make* была разработана для того, чтобы автоматизировать рутинные задачи трансформации исходного кода в исполняемые файлы. Преимущество *make* над простыми сценариями командного интерпретатора состоит в возможности описать отношения между элементами проекта. *make*, основываясь на описанных отношениях и данных о времени модификации файлов, сможет определить, какие именно шаги необходимо осуществить для получения необходимой вам версии программы. Используя эту информацию, *make* также сможет оптимизировать процесс сборки и избежать выполнения ненужных действий.

GNU *make* предоставляет язык для описания отношений между исходным кодом, промежуточными и исполняемыми файлами. *make* также включает функциональность для управления конфигурациями, реализации библиотек спецификаций, пригодных для повторного использования, и параметризации процесса сборки через механизм макросов, определяемых пользователем. *make* может рассматриваться как каркас всего процесса разработки, выделяющий компоненты приложе-

ния и описывающий способы связать их в единое целое.

Спецификация, используемая *make*, обычно сохраняется в файле с именем *makefile*. Ниже приведён пример *makefile*'а пригодного для сборки традиционной программы «Hello, World»:

```
hello: hello.c
    gcc hello.c -o hello
```

Для того, чтобы собрать программу, достаточно выполнить в командном интерпретаторе следующую команду:

```
$ make
```

Это приведёт к запуску *make*, который затем прочитает *makefile* и соберёт первую цель, определённую в нём. В итоге вы увидите следующее:

```
$ make
gcc hello.c -o hello
```

Цель может быть передана в качестве аргумента командной строки, в этом случае *make* будет пытаться собрать указанную вами цель. В противном случае для сборки будет выбрана первая цель, определённая в *makefile*'е (называемая также *целью по умолчанию*).

Обычно цель по умолчанию предназначена для сборки всего приложения, и процесс сборки этой цели состоит из определённой последовательности шагов. Например, довольно часто исходный код приложения должен быть составлен при помощи программ наподобие *flex* или *bison*. Полученный исходный код нужно скомпилировать в объектные файлы (файлы с расширением *.o* или *.obj* для C/C++ и *.class* для Java). Затем объектные файлы (для случая C/C++) должны быть собраны компоновщиком (обычно вызываемым компилятором *gcc*) для получения исполняемого файла.

Модификация любого из исходных файлов требует повторного вызова *make*, который инициирует повторение некоторых (обычно не всех) из вышеописанных действий таким образом, чтобы получить исполняемый файл с новой функциональностью. Файл спецификаций, *makefile*, описывает отношения между исходным кодом, промежуточными и исполняемыми файлами, что позволяет *make* выполнять минимум необходимой работы для получения новой версии исполняемого файла.

Принципиальное значение *make* заключается в его возможности осуществлять сложные последовательности операций, необходимые для сборки приложения, и производить оптимизацию выполнения этих операций для максимального сокращения времени, занимаемого циклом «модификация-компиляция-отладка». Более того, этот инструмент настолько гибок, что может быть использован везде, где

существуют зависимости между файлами, начиная с традиционных программ на C/C++ и заканчивая приложениями на Java, документами TEX, управлением базами данных и конвертацией изображений.

1.1 Цели и реквизиты

По существу *makefile* содержит набор правил для сборки приложения. Первое правило, обнаруженное *make*, становится *правилом по умолчанию*. Правила состоят из трёх составляющих: целей, реквизитов и сценария сборки:

```
цель: реквизит1 реквизит2
    сценарий сборки
```

Цель — это файл или некоторая сущность, требующая обновления. *Реквизиты* — это те файлы, которые должны существовать для того, чтобы цель могла быть собрана. *Сценарий сборки* — это команды интерпретатора, описывающие способ создания цели из реквизитов.

Вот правила компиляции исходного файла на языке C *foo.c* в объектный файл *foo.o*:

```
foo.o: foo.c foo.h
    gcc -c foo.c
```

Цель *foo.o* указана слева от двоеточия, реквизиты *foo.c* и *foo.h* — справа. Сценарий сборки обычно начинается со следующей строки и предваряется символом табуляции.

Когда требуется выполнить правило, *make* пытается найти все файлы-реквизиты, ассоциированные с целью. Если хотя-бы один из реквизитов выступает в качестве цели какого-либо правила, то сначала выполняется правило для этого реквизита. Далее проверяются даты последней модификации файла-цели и реквизитов: если какой-либо из реквизитов модифицировался позже цели, то выполняется сценарий сборки. Каждая строка сценария выполняется в отдельном процессе командного интерпретатора. Если хотя-бы одна команда приводит к ошибке, сборка цели завершается и *make* прекращает выполнение.

Ниже представлена программа подсчёта числа вхождений слов «fee», «fie», «foe» и «fum» в содержимое стандартного потока ввода, реализованная с помощью лексического анализатора *flex*.

```
#include <stdio.h>

extern int fee_count, fie_count, foe_count, fum_count;
```

```
extern int yylex( void );

int main( int argc, char ** argv )
{
    yylex( );

    printf( "%d %d %d %d \n",
        fee_count, fie_count, foe_count, fum_count );

    exit( 0 );
}
```

Код лексического анализатора очень прост:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
```

Составление *makefile* для сборки этой программы также не вызывает затруднений:

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -o count_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

Когда мы запустим *make* впервые, мы увидим следующее:

```
$ make

gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc counti_words.o lexer.o -lfl -ocount_words
```

Теперь мы имеем исполняемый файл программы. Естественно, реальные приложения, как правило, состоят из большего числа модулей, чем наш пример. Кроме того, позже мы увидим, что наш *makefile* не использует большей части функциональности *make* и потому гораздо более объёмен чем мог бы быть. Тем не менее, это всё-же функциональный и полезный *makefile*.

Как вы могли заметить, порядок, в котором команды реально выполняются, и порядок их появления в *makefile*'е противоположны. Стиль опеделения «сверху вниз» является общим стилем составления спецификаций для *make*. Обычно сначала описываются правила для наиболее важных целей, а детали оставляются на потом. В *make* существует несколько механизмов поддержки такого стиля. Главными среди них являются двухфазовая модель выполнения *make* и рекурсивные переменные. Мы обсудим эти важные детали в последующих главах.

1.2 Разрешение зависимостей

Как же *make* решает, что делать? Давайте рассмотрим выполнение предыдущего примера более детально и выясним это.

Сначала *make* замечает, что командная строка не содержит аргументов и решает собрать цель по умолчанию, т.е. *count_words*. Затем выполняется проверка реквизитов, их обнаруживается три: *count_words.o*, *lexer.o* и *-lfl*. Затем *make* ищет способ собрать цель *count_words.o* и находит правило для неё. Снова происходит проверка реквизитов. *make* замечает, что цель *count_words.c* не имеет правила, и файл с таким именем существует, поэтому производится трансформация *count_words.c* в *count_words.o*, для осуществления которой выполняется следующая команда:

```
gcc -c count_words.c
```

Подобная цепь переходов от целей к реквизитам и рассмотрение реквизитов в качестве целей — типичный механизм, при помощи которого *make* проводит анализ *makefile*'а и решает, какие команды подлежат выполнению.

Следующим реквизитом, рассматриваемым *make*, является *lexer.o*. Цепочка правил, подобная рассмотренной, ведёт к файлу *lexer.c*, ещё не существующему. Далее *make* находит правило получения *lexer.c* из *lexer.l*, согласно которому запускается программа *flex*. Теперь *lexer.c* существует и можно запускать *gcc*.

Наконец, *make* проверяет реквизит *-lfl*. Опция `-l` сообщает *gcc*, что приложение использует системную библиотеку. Имя библиотеки «fl» преобразуется согласно правилам именования библиотек в *libfl.a*. GNU *make* имеет поддержку этого синтаксиса. Когда находится реквизит вида *-l<NAME>*, *make* ищет файл с именем *libNAME.so* в стандартных каталогах библиотек. Если поиск заканчивается неудачей, производится повторный поиск по имени *libNAME.a*. В нашем случае поиск

успешен, *make* находит файл */usr/lib/libfl.a* и производит финальное действие — компоновку.

1.3 Минимизируем число действий

Если мы запустим нашу программу, то обнаружим, что помимо вывода числа вхождений слов fee, fies, foes и fums она также выводит содержимое входного файла. Это совсем не то, что от неё ожидалось. Проблема в том, что мы забыли добавить несколько правил в наш лексический анализатор, и *flex* выдаёт нераспознанный текст на стандартный поток вывода. Для решения этой проблемы мы просто добавим правило для «любого символа»:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
.
\n
```

После редактирования файла с исходным кодом анализатора нам нужно собрать наше приложение заново и проверить его работу:

```
$ make

flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

На этот раз файл *count_ words.c* не подвергся компиляции. При анализе правил *make* обнаружил, что файл *count_ words.o* существует и имеет более позднюю дату модификации, чем цель, потому не было предпринято никаких действий по сборке этой цели. Однако при анализе цели *lexer.c* было обнаружено, что реквизит *lexer.l* имеет более позднюю дату модификации, поэтому произошла повторная сборка цели *lexer.c*. Это, в свою очередь, вызвало повторную сборку *lexer.o*, а затем и *count_ words*. Теперь наша программа подсчёта слов работает правильно:

```
$ count_words < lexer.l
3 3 3 3
```

1.4 Вызов *make*

В предыдущих примерах предполагается, что:

- Все исходные файлы проекта и файл спецификации *make* хранятся в одной директории.
- Файл спецификации для *make* называется *makefile*, *Makefile* или *GNUMakefile*.
- *makefile* находится в текущей директории, когда пользователь запускает команду *make*.

Если *make* запускается с соблюдением этих условий, автоматически производится попытка собрать первую цель в файле спецификации. Чтобы собрать другую цель (или несколько целей), необходимо указать имя этой цели в качестве аргументов командной строки:

```
$ make lexer.c
```

В этом случае после запуска *make* произведёт чтение файла спецификации и определение цели для обновления. Если цель или один из реквизитов устарели (или не существуют), тогда в точности один раз будет выполнен сценарий их сборки. После выполнения сценария цель предполагается существующей и обновлённой, и процесс повторяется для следующей цели; после сборки всех целей процесс завершается.

Если указанная вами цель не требует повторной сборки, *make* завершит работу с сообщением следующего вида:

```
$ make lexer.c
make:  'lexer.c' is up to date.
```

Если в качестве цели будет указана цель, не присутствующая в *makefile*'е, и для которой не существует неявного правила (см. главу 2), то *make* закончит выполнение с сообщением следующего вида:

```
$ make non-existent-target
make: *** No rule to make target 'non-existent-target'. Stop.
```

У команды *make* есть множество опций командной строки. Одной из самых полезных является опция `--just-print` (или просто `-n`), которая сообщает *make*, что нужно только отобразить последовательность действий, необходимых для сборки цели. Эту возможность очень удобно использовать для отладки *makefile*'ов. Также полезной возможностью является передача или переопределение значений переменных *make* через аргументы командной строки.

1.5 Основы синтаксиса *makefile*

Теперь, когда вы уже получили базовое представление о *make*, можно приступить к составлению собственных *makefile*'ов. В этом разделе мы рассмотрим синтаксис и структуру *makefile*'а, чтобы вы могли начать использование *make*.

Как правило, *makefile*'ы пишутся в манере «сверху вниз», то есть сначала описывается наиболее общая цель (как правило, она имеет имя `all`), которая будет целью по умолчанию. Далее следуют всё более детализированные цели, и в самом конце описываются цели, используемые для поддержки программного продукта (такие, например, как `clean`, используемая для удаления ненужных вр'еменных файлов). Именами целей вовсе не обязательно должны быть настоящие файлы, можно использовать любое имя.

В предыдущем примере мы видели упрощённую форму правила. Более полной (но всё ещё не законченной) формой правила является следующая:

цель$_1$ *цель*$_2$ *цель*$_3$ *: реквизит*$_1$ *реквизит*$_2$
    команда$_1$
    команда$_2$
    команда$_3$

Одна или более целей указываются слева от двоеточия, справа от него следуют реквизиты. Если реквизиты не указаны, то собираются только те цели, которые ещё не существуют. Последовательность команд, которые необходимо выполнить для сборки цели, иногда называют *сценарием сборки*, но чаще просто *командами*.

Каждая команда должна начинаться с символа табуляции. Такой синтаксис сообщает *make* о том, что следующие за табуляцией символы должны быть переданы в командный интерпретатор для последующего выполнения. Если вы случайно поставите символ табуляции в начале строки, не являющейся командой, то в большинстве случаев *make* будет интерпретировать последующий текст в этой строке как команду. Однако может случиться и так, что *make* сможет распознать ваш символ табуляции как синтаксическую ошибку, в этом случае вы увидите сообщение, подобное следующему:

```
$ make
Makefile:6: *** commands commence before first target.  Stop.
```

Мы вернёмся к аспектам использования символа табуляции в главе 2.

Глава 2

Правила

В предыдущей главе мы рассмотрели несколько правил для компиляции и компоновки программы подсчёта слов. Каждое из этих правил определяло цель — файл, который требуется собрать. Каждая цель зависела от множества реквизитов, которые также являлись файлами. Когда требовалось обновить цель, *make* выполнял сценарий сборки только в том случае, если файлы реквизитов имели дату модификации более позднюю, чем цель. Поскольку цель одного правила может быть реквизитом другого, множество целей и реквизитов может быть представлено в форме *графа зависимостей* (*dependency graph*). Составление и обработка графа зависимостей является основной работой *make*.

Поскольку правила так важны для *make*, существует несколько их разновидностей. *Явные правила*, наподобие тех, что были использованы в предыдущей главе, указывают на необходимость обновления цели при модификации или отсутствии файлов-реквизитов. Правила этого типа вы будете писать наиболее часто. *Шаблонные правила* используют подстановки (wildcards) вместо явного указания имён файлов. Это позволяет *make* применять такие правила каждый раз при соответствии имени цели некоторому шаблону. *Неявные правила* — это явные или суффиксные правила, встроенные в базу данных правил *make*. Наличие встроенной базы данных правил упрощает написание *makefile*'ов, поскольку для многих общих задач уже известны типы файлов, суффиксы и сценарии сборки целей. *Статические шаблонные правила* отличаются от обычных шаблонных правил тем, что могут быть применены только к определённому списку целей.

GNU *make* может быть использован как замена для многих других версий *make*. Он включает в себя множество возможностей, сохранённых для поддержания обратной совместимости. Например, *Суффиксные правила* были реализованы в одной из первых версий *make* для написания общих правил. *make* имеет поддержку сиффиксных правил, однако они признаны устаревшими, поскольку могут быть заменены более простыми и более óбщными шаблонными правилами.

2.1 Явные правила

Чаще всего вам придётся писать именно явные правила, указывающие некоторые файлы как цели и реквизиты. Правило может иметь более одной цели. Это значит, что каждая из указанных целей имеет в точности то же множество реквизитов, что и остальные цели. Если цели требуется обновить, для каждой из них будет выполнен один и тот же сценарий. Вот пример такого правила:

```
vpath.o variable.o: make.h config.h getopt.h gettext.h dep.h
```

Это правило означает, что цели *vpath.o* и *variable.o* зависят от одного и того же множества заголовочных файлов. Оно имеет в точности тот же эффект, что и следующая спецификация:

```
vpath.o: make.h config.h getopt.h gettext.h dep.h

variable.o: make.h config.h getopt.h gettext.h dep.h
```

Обе цели собираются независимо. Если один из объектных файлов имеет более раннюю дату модификации, чем один из указанных заголовочных файлов, *make* инициирует сборку и выполнит команды, ассоциированные с правилом.

Правило не обязательно указывать полностью сразу. Каждый раз, когда *make* обнаруживает файл в качестве цели, он добавляет цель и реквизиты в граф зависимостей. Если такая цель уже существовала в графе, к записи о цели добавляются новые реквизиты. Одним из элементарных применений этого свойства является разбиение длинной строки на несколько более коротких для улучшения читабельности файла:

```
vpath.o: make.h config.h getopt.h gettext.h dep.h
vpath.o: filedef.h hash.h job.h commands.h variable.h vpath.h
```

В наиболее сложных случаях список реквизитов может состоять из файлов, способы обработки которых различны:

```
# Убедимся, что файл lexer.c существует до компиляции vpath.c
vpath.o: lexer.c

...

# Компилируем vpath.c с определёнными флагами
vpath.o: vpath.c
$(COMPILE.c) $(RULE_FLAGS) $(OUTPUT_OPTION) $<

...

# Включаем файл зависимостей, составленный программой
include auto-generated-dependencies.d
```

Первое правило декларирует, что цель *vpath.o* должна быть собрана заново при изменении файла *lexer.c* (возможно, генерация этого файла имеет некий побочный эффект). Правило также может быть использовано для того, чтобы убедиться, что все реквизиты существуют (или, в случае необходимости, обновлены) перед сборкой цели. Нужно отметить двустороннюю сущность правил. При прямом чтении правило означает, что, если файл *lexer.c* изменился, требуется выполнить действия по обновлению *vpath.o*. При чтении в обратном направлении правило означает, что если требуется обновить *vpath.o*, то нужно убедиться, что файл *lexer.c* существует. Это правило может быть помещено рядом с остальными правилами, касающимися файла *lexer.c*, чтобы разработчики помнили об этой тонкой взаимосвязи. Далее, рассмотрим правило компиляции *vpath.o*. Сценарий сборки для этого правила использует три переменных *make*. Переменные будут детально описаны позже, пока важно знать лишь то, что обращение к переменной происходит с помощью знака доллара (`$`), за которым следует либо один символ, либо слово в круглых скобках. Наконец, зависимости типа *.o/.h* включаются из отдельного файла, полученного при помощи внешней программы.

В качестве особого случая GNU *make* поддерживает упрощённый синтаксис для правил с одной командой.

```
цель: ; команда
```

На практике такие правила встречаются редко, однако всё же иногда они могут быть полезны, особенно когда нужно сберечь место на экране монитора или листе бумаги.

2.1.1 Шаблоны

Часто *makefile*'ы содержат огромное количество файлов. Для упрощения работы с ними *make* поддерживает шаблоны, идентичные шаблонам командного интерпретатора *Bourne shell*: `~`, `*`, `?`, `[...]` и `[^...]`. Например, шаблону `*.*` соответствуют все файлы, содержащие в имени точку. Знак вопроса означает один символ, а `[...]` — класс символов. Для выбора дополнения класса символов нужно использовать `[^...]`. Знак тильды (`~`) может быть использован для обозначения домашнего каталога текущего пользователя системы. Если за тильдой следует имя пользователя, будет подставлен домашний каталог указанного пользователя. *make* автоматически раскрывает шаблоны, когда они встречаются в названиях целей, реквизитов или командных сценариях. В другом контексте шаблоны могут быть раскрыты явным вызовом функции. Шаблоны чрезвычайно полезны для написания более адаптивных *makefile*'ов. Например, вместо того, чтобы явно перечислять все файлы, входящие в состав исходного кода программы, вы можете использовать

шаблоны[1]:

```
prog: *.c
$(CC) -o $@ $^
```

Однако очень важно быть осторожным с шаблонами, ими легко злоупотребить. Рассмотрим пример:

```
*.o: constants.h
```

Намерения очевидны: все объектные файлы зависят от заголовочного файла *constants.h*. Однако посмотрим, как раскроется шаблон в каталоге, не содержащем объектных файлов:

```
: constants.h
```

Это допустимое выражение *make*, оно не вызовет ошибки, однако оно также не выразит той зависимости, которую имел в виду пользователь. Одним из корректных способов реализации этого правила является использование шаблона для получения файлов с исходным кодом (которые, как правило, присутствуют) и трансформация полученного списка в список объектных файлов. Мы рассмотрим эту технику при обсуждении функций в главе 4.

Наконец, стоит отметить, что раскрытие шаблонов в тот момент, когда они появляются в качестве целей или реквизитов, осуществляет непосредственно *make*. Однако раскрытие шаблонов в сценариях происходит в дочернем процессе командного интерпретатора. Это может быть важной деталью, поскольку *make* раскрывает шаблоны во время чтения *makefile*'а, а командный интерпретатор раскрывает их много позже, во время непосредственного выполнения команд. Когда производятся сложные манипуляции с файлами, результаты раскрытия одинаковых шаблонов в разные моменты времени могут сильно отличаться. Проблематичной может быть ситуация, когда некоторые файлы являются результатом сборки, и *make* не видит их во время обработки *makefile*'а. К таким случаям нужно относится особенно осторожно.

2.1.2 Абстрактные цели

До этого момента все цели и реквизиты, рассматриваемые нами, были файлами, которые нужно было создать или обновить. Хоть это и типичный способ использования целей, часто бывает полезным представлять цель в качестве метки

[1] В более серьёзных приложениях применение шаблонов для выбора компилируемых файлов является плохой практикой, поскольку может вызвать компоновку с посторонним опасным кодом. В правилах удаления промежуточных файлов шаблоны могут быть фатальными для проекта (прим. автора).

для командного сценария. Например, ранее упоминалось, что стандартной целью для многих *makefile*'ов является `all`. Цели, не представляющие файлов, называют *абстрактными целями* (*phony targets*). Ещё одной стандартной абстрактной целью является `clean`:

```
clean:
    rm -f *.o lexer.c
```

Абстрактные цели должны собираться всегда, потому что команды, ассоциированные с правилом, не создают файл с именем цели.

Важно заметить, что *make* не отличает абстрактных целей от целей, являющимися файлами. Если по случайности файл с именем абстрактной цели существует, *make* будет ассоциировать этот файл с абстрактной целью в графе зависимостей. Например, если в текущей директории существует файл *clean*, то запуск команды `make clean` приведёт к появлению довольно неожиданного сообщения:

```
$ make clean
make: ‘clean’ is up to date.
```

Довольно часто абстрактные цели не имеют реквизитов; цель `clean` всегда будет рассматриваться как не требующая обновления, и ассоциированные с ней команды никогда не будут выполнены.

Чтобы избежать этой проблемы, GNU *make* имеет специальную цель, `.PHONY`, позволяющую сообщить *make*, что цель не является настоящим файлом. Любая цель может быть объявлена как абстрактная с путём включения её в список реквизитов цели `.PHONY`:

```
.PHONY: clean
clean:
    rm -f *.o lexer.c
```

Теперь *make* всегда будет выполнять команды, ассоциированные с целью `clean`, даже если файл с таким именем существует. В добавок к пометке цели как требующей обновления, спецификация цели как абстрактной сообщает *make*, для этой цели не нужно использовать стандартное правило получения файла цели из исходного кода. Это позволяет *make* провести оптимизацию обычного процесса поиска правил для достижения более высокой производительности.

Довольно редко имеет смысл включать абстрактную цель в качестве реквизита реального файла, поскольку это будет приводить к безусловному обновлению цели. Указание же реквизитов абстрактных целей довольно часто приносит пользу. Например, цель `all` имеет в качестве реквизитов список программ, которые нужно собрать:

```
.PHONY: all
all: bash bashbug
```

В предыдущем примере цель `all` собирает командный интерпретатор *bash* и инструмент отправки сообщений об ошибках *bashbug*.

Абстрактные цели могут рассматриваться как сценарии интерпретатора, встроенные в *makefile*. Объявление абстрактной цели в качестве реквизита другой цели вызовет запуск сценария, ассоциированного с абстрактной целью, перед сборкой основной цели. Предположим, мы ограничены в использовании дискового пространства, и хотим отобразить количество доступного места на диске перед выполнением действий, требующих значительных затрат дискового пространства. Одно из решений демонстрирует следующий пример:

```
.PHONY: make-documentation
make-documentation:
    df -k . | awk 'NR == 2 { printf( "%d available\n", $$4 ) }'
    javadoc ...
```

Проблема заключается в том, что нам может понадобиться указать команды *df* и *awk* несколько раз для разных целей. Это является проблемой с точки зрения поддержки, поскольку нам придётся изменять каждое вхождение этих команд, если, например, в другой системе формат выдачи данных утилиты *df* отличается. Поэтому более изящным решением является следующее:

```
.PHONY: make-documentation
make-documentation: df
    javadoc ...

.PHONY: df
df:
    df -k . | awk 'NR == 2 { printf( "%d available\n", \$\$4 ) }'
```

Мы можем сообщить *make* о необходимости вызова сценария, ассоциированного с целью *df*, перед созданием документации, указав *df* как реквизит цели `make-documentation`. Это допустимо, поскольку `make-documentation` также является абстрактной целью. Такой подход даёт нам ещё одно преимущество: теперь мы можем легко использовать *df* в других целях.

Существует много примеров удачного применения абстрактных целей.

Сообщения *make* довольно трудны для чтения и отладки. На это есть несколько причин: составление *makefile*'ов сверху-вниз, в то время как команды выполняются снизу-вверх; кроме того, не указывается, какая цель выполняется в данный момент. Чтобы исправить ситуацию, полезно выводить сообщения о начале выполнения основных целей. Абстрактные цели являются простым средством реализации этой идеи. Ниже приведён отрывок из *makefile*'а командного интерпретатора *bash*:

Цель	**Назначение**
`all`	Произвести сборку приложения.
`install`	Произвести установку собранного приложения.
`clean`	Удалить все бинарные файлы, полученные после сборки.
`distclean`	Удалить все файлы, не входящие в базовый дистрибутив.
`TAGS`	Создать таблицу тэгов для текстового редактора.
`info`	Создать файлы GNU info из файлов Texinfo.
`check`	Запустить все тесты, ассоциированные с приложением.

Таблица 2.1: Стандартные абстрактные цели.

```
$(Program): build_msg $(OBJECTS) $(BUILTINS_DEP) $(LIBDEP)
    $(RM) $@
    $(CC) $(LDFLAGS) -o $(Program) $(OBJECTS) $(LIBS)
    ls -l $(Program)
    size $(Program)

.PHONY: build_msg
build\_msg:
    @printf "#\n# Building $(Program)\n#\n"
```

Поскольку `build_msg` является абстрактной целью, сообщение выводится непосредственно перед проверкой остальных реквизитов. Если бы сообщение о начале сборки было первой командой сценария сборки `$(Program)`, то оно выводилось бы только после сборки всех зависимых файлов. Также важно заметить, что, поскольку абстрактные цели всегда помечены как требующие обновления, указание абстрактной цели `build_msg` в качестве реквизита `$(Program)` вызовет безусловную сборку этой цели, даже если на самом деле этого не требуется. В нашем случае это выглядит разумным, поскольку основная работа заключается в компиляции исходных файлов в объектные, а на финальном этапе будет производиться только компоновка.

Абстрактные цели также могут быть использованы для улучшения «пользовательского интерфейса» *makefile*'а. Имена целей часто содержат длинные стоки с путями к каталогам, дополнительные имена компонентов (например, номера версий) и стандартные суффиксы. Это может сделать указание имени нужной цели довольно неудобным занятием. Этой проблемы можно избежать, добавив абстрактную цель указав в качестве её реквизита имя нужной реальной цели.

Есть ряд абстрактных целей, являющихся более или менее стандартными. Не смотря на то, что их имена являются лишь соглашением, эти цели встречаются в большинстве *makefile*'ов. Список этих целей содержится в Таблице 2.1.

Цель `TAGS` на самом деле не является абстрактной, поскольку программы *ctags* и *etags* создают файл с именем *TAGS*. Эта цель включена в таблицу потому, что

это единственная стандартная реальная цель.

2.1.3 Пустые цели

Пустые цели подобны абстрактным в том плане, что позволяют расширить возможности *make*. Абстрактные цели всегда требуют обновления и вызывают сборку всех целей, *зависимых* от абстрактной (т.е. содержащих её в списке реквизитов). Предположим, однако, что у нас есть команда, не ассоциированная с файлом, которую нужно выполнять время от времени, причём зависимые цели не должны при этом обновляться. Для этого мы можем воспользоваться целью, ассоциированной с пустым файлом:

```
prog: size prog.o
    $(CC) $(LDFLAGS) -o $@ $^

size: prog.o
    size $^
    touch size
```

Заметим, что правило `size` использует программу *touch* после своего завершения. Пустой файл используется только для хранения времени последней модификации, и *make* будет выполнять правило `size` только в том случае, если файл *prog.o* подвергся изменению. Более того, спецификация `size` как реквизита *prog* будет вызывать обновление `prog` только в том случае, если соответствующий объектный файл изменялся.

Пустые файлы бывают полезны в сочетании с автоматической переменной `$?`. Мы обсудим автоматические переменные в разделе «Автоматические переменные», но краткое описание этой переменной здесь не повредит. Внутри сценария сборки каждого правила *make* определяет переменную `$?` как множество реквизитов, имеющих более позднюю дату модификации, чем цель. Вот пример правила, печатающего имена всех файлов, изменившихся с момента последнего выполнения команды `make print`:

```
print: *.[hc]
    lpr $?
    touch $@
```

2.2 Переменные

Рассмотрим некоторые из тех переменных, которые мы использовали в наших примерах. Самые простые из них имели следующий синтаксис:

$(имя-переменной)

Эта запись означает, что мы хотим получить значение переменной с именем `имя-переменной`. Переменные могут содержать практически произвольный текст, а имена переменных допускают использование большинства символов, включая знаки пунктуации. Например, переменная, содержащая имя команды для компиляции исходного кода на языке C, имеет имя `COMPILE.c`. Как правило, для подстановки значения переменной её имя окружают символами `$(` и `)`. В случае, когда имя переменной состоит из одного символа, скобки можно опускать.

Как правило, *makefile*'ы содержат много объявлений переменных. Кроме того, существует множество переменных, определяемых непосредственно *make*. Некоторые из них предназначены для контроля пользователем поведения *make*, другие выставляются *make* для взаимодействия с пользовательским *makefile*'ом.

Автоматические переменные

Автоматические переменные вычисляются *make* заново для каждого исполняемого правила. Они предоставляют доступ к цели и списку реквизитов, избавляя от необходимости явно указывать имена файлов. Автоматические переменные полезны для избежания дублирования кода и необходимы для написания шаблонных правил (их мы рассмотрим позже).

Существует семь автоматических переменных:

`$@`
: Имя файла цели правила. Если цель является элементом архива (archive member), то «`$@`» обозначает имя файла архива.

`$%`
: Для целей, являющихся элементами архива, обозначает имя элемента. Если цель не является элементом архива, то `$%` содержит пустое значение.

`$<`
: Имя первого реквизита в списке реквизитов.

`$?`
: Имена всех реквизитов, имеющих более позднюю дату модификации, чем цель.

`$^`
: Имена всех реквизитов, разделённые пробелами. В списке отсутствуют повторения элементов, поскольку для большинства задач (копирование, компиляция и т.д.) повторения нежелательны.

`$+`
: Подобно `$?`, содержит список имён реквизитов, разделённых пробелами, с

тем отличием, что может содержать повторения. Эта переменная была введена для специфических ситуаций, таких как компоновка, где повторение аргументов несёт особый смысл.

`$*`

Основа имени файла цели. Как правило, основой является имя файла с отброшенным суффиксом (мы рассмотрим вычисление основы в разделе «Шаблонные правила»). Использование этой переменной вне шаблонных правил настоятельно не рекомендуется.

Кроме того, каждая из вышеперечисленных переменных имеет два варианта для совместимости с другими версиями *make*. Один из этих вариантов возвращает название каталога, в которой находится соответствующий файл. Этот вариант обозначается добавлением символа «D» к имени переменной: `$(@D)`, `$(<D)` и т.д. Второй вариант возвращает только имена файлов без имени каталога, в котором они находятся. Этот вариант обозначается добавлением символа «F» к имени переменной: `$(@F)`, `$(<F)` и т.д. Поскольку эти варианты имён содержат более одного символа, они должны заключаться в круглые скобки. GNU *make* предоставляет более читабельные альтернативы в лице функций *dir* и *notdir*. Мы обсудим эти функции в главе 4.

Вот пример нашего *makefile*'а, в котором явно указанные имена заменены подходящими автоматическими переменными.

```
count_words: count_words.o counter.o lexer.o -lfl
    gcc $^ -o $@

count_words.o: count_words.c
    gcc -c $<

counter.o: counter.c
    gcc -c $<

lexer.o: lexer.c
    gcc -c $<

lexer.c: lexer.l
    flex -t $< > $@
```

2.3 Поиск файлов с помощью VPATH и vpath

Наши примеры были довольно просты, поэтому *makefile* и исходные файлы находились в одном каталоге. Реальные программы гораздо сложнее (когда в последний раз вы работали над проектом, все файлы которого размещались в одном каталоге?). Давайте изменим наш пример, разместив файлы более реалистичным

образом. Мы может модифицировать нашу программу подсчёта слов, перенеся часть работы, выполняемой функцией *main*, в функцию *counter*:

```
#include <lexer.h>
#include <counter.h>

void counter( int counts[4] )
{
    while ( yylex( ) )
        ;

    counts[0] = fee_count;
    counts[1] = fie_count;
    counts[2] = foe_count;
    counts[3] = fum_count;
}
```

Поместим объявление этой функции в заголовочный файл *counter.h*:

```
#ifdef COUNTER_H_
#define COUNTER_H_

extern void
counter( int counts[4] );

#endif
```

Мы также можем поместить объявления функций из *lexer.l* в файл *lexer.h*:

```
#ifndef LEXER_H_
#define LEXER_H_

extern int fee_count, fie_count, foe_count, fum_count;

extern int yylex( void );

#endif
```

В соответствии с негласным правилами размещения исходного кода в файловой системе, все заголовочные файлы помещаются в каталог *include*, а исходные файлы — в каталог *src*. Разместим наши файлы подобным образом, оставив *makefile* в корневом каталоге проекта. Результирующее дерево изображено на рисунке 2.1.

Поскольку теперь наши файлы с исходным кодом включают заголовочные файлы, эта новая зависимость должна быть отражена в нашем *makefile*, то есть если модифицируется заголовочный файл, то соответствующий объектный файл должен быть заново скомпилирован из исходного кода.

```
.
|--include
|  |--counter.h
|  `--lexer.h
|--src
|  |--count\_words.c
|  |--counter.c
|  `--lexer.l
`--Makefile
```

Рис. 2.1: Структура каталогов проекта программы подсчёта слов.

```
count_words: count_words.o counter.o lexer.o -lfl
    gcc $^ -o $@

count_words.o: count_words.c include/counter.h
    gcc -c $<

counter.o: counter.c include/counter.h include/lexer.h
    gcc -c $<

lexer.o: lexer.c include/lexer.h
    gcc -c $<

lexer.c: lexer.l
    flex -t $< > $@
```

После запуска *make* получаем следующий результат:

```
$ make
make:  *** No rule to make target `count_words.c',
       needed by `count_words.o'.  Stop.
```

Что же произошло? *make* попытался собрать исходный файл *count_ words.c*! Давайте проследим за работой *make*. Первым реквизитом является файл *count_ words.o*. Такой файл не существует, следовательно, его нужно создать. Явное правило для создания *count_ words.o* указывает на *count_ words.c*. Но *make* не сможет найти этот исходный файл, ведь он находится не в текущем каталоге, а в каталоге *src*. Пока вы не укажите другого, *make* будет искать цели и реквизиты в текущем каталоге. Как сообщить *make*, что исходные файлы нужно искать в каталоге *src*, или, если поставить вопрос более óбщно, где нужно искать файлы с исходным кодом?

Вы можете указать *make* каталоги, в которых нужно искать файлы, используя переменную `VPATH` и директиву `vpath`. Для того, чтобы решить нашу проблему, мы может добавить определение `VPATH` в наш *makefile*:

```
VPATH = src
```

Это означает, что *make* должен искать в каталоге *src* те файлы, которые не обнаружены в текущем каталоге. После запустка *make*, мы получим следующий вывод:

```
$ make
gcc -c src/count_words.c -o count_words.o
src/count_words.c:2:21: counter.h: No such file or directory
make: *** [count_words.o] Error 1
```

Заметим, что в этот раз *make* успешно запустил команду компиляции, правильно подставив относительный путь к исходному файлу. В этом заключается ещё один плюс использования автоматических переменных: *make* не сможет использовать подходящий путь к исходному файлу, если вы явно укажите его имя. К несчастью, процесс компиляции окончился неудачей, так как *gcc* не смог найти заголовочного файла. Эту проблему можно решить, настроив неявное правило компиляции подходящей опцией `-I`:

```
CPPFLAGS = -I include
```

Теперь сборка проходит успешно:

```
$ make

gcc -I include -c src/count_words.c -o count_words.o
gcc -I include -c src/counter.c -o counter.o
flex -t src/lexer.l > lexer.c
gcc -I include -c lexer.c -o lexer.o
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
```

Переменная `VPATH` содержит список всех каталогов, в которых *make* будет искать недостающие файлы. В этих каталогах будет происходить поиск как целей, так и реквизитов, но не файлов, указанных в сценариях сборки. Элементы списка могут разделяться пробелами или двоеточиями в Unix и пробелами или точками с запятой в Windows. Предпочтительней пользоваться пробелами, поскольку такой формат подходит для любой операционной системы и позволяет избежать путаницы с точками с запятой и двоеточиями. Кроме того, имена каталогов, разделённые пробелами, легче читать.

Переменная `VAPTH` решает нашу проблему поиска, но всё же она является слишком мощным инструментом. *make* ищет в каждом указанном в `VPATH` каталоге *каждый* недостающий файл. Если в разных каталогах существуют файлы с одинаковыми именами, *make* возьмёт первый найденный. Иногда это может быть причиной неприятностей.

Директива `vpath` больше подходит для наших нужд. Синтаксис вызова этой директивы представлен ниже:

```
vpath шаблон список-каталогов
```

Теперь мы можем исключить использование переменной `VPATH`, добавив следующие строки в наш *makefile*:

```
vpath %.c src
vpath %.h include
```

Теперь мы сообщили *make*, что искать все файлы с расширением *.c* нужно в каталоге *src*, а файлы с расширением *.h* — в каталоге *include* (теперь мы можем удалить префикс *include* у заголовочных файлов в списке реквизитов). В более сложных приложениях такой подход позволяет сохранить нервы и время, потраченные на отладку.

В нашем примере мы использовали `vpath` для решения проблемы поиска файлов, разнесённых по разным директориям. Существует также родственная проблема сборки приложения таким образом, чтобы объектные файлы помещались в отдельное «бинарное» дерево каталогов, в то время как исходные файлы хранились дереве каталогов «исходного кода». Правильное использование `vpath` помогает решить и эту проблему, однако эта задача быстро становиться очень сложной, так что использование одной лишь директивы `vpath` становится неэффективным. Мы обсудим эту проблему более детально в следующих разделах.

2.4 Шаблонные правила

Все примеры *makefile*'ов, рассмотренные нами, были слишком подробны. Для маленькой программы, содержащей десяток файлов это не имеет большого значения, но явная спецификация всех целей, реквизитов и сценариев сборки для программы, содержащей сотни и тысячи файлов, не представляется возможным. Более того, команды сами по себе вносят дублирующийся код в наш *makefile*. Это может быть источником ошибок и большой проблемой при поддержке.

Многие программы, читающие один тип файлов и выводящие другой следуют стандартным соглашениям. Например, все компиляторы языка C предполагают, что файлы с расширением *.c* содержат исходный код на языке C, а имена объектных файлов могут быть получены заменой расширения *.c* на расширение *.o* (или *.obj* для некоторых компиляторов в Windows). В предыдущей главе мы увидели, что исходные файлы генератора *flex* имеют расширение *.l*, а расширением генерируемых им файлов является *.c*.

Эти и другие соглашения позволяют *make* упростить создание правил с помощью распознавания шаблонов и предоставления встроенных правил сборки. Например, используя встроенные правила, мы можем сократить наш *makefile* с семнадцати строк до шести:

```
VPATH = src include
CPPFLAGS = -I include

count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
```

Все встроенные правила являются шаблонными. Шаблонные правила похожи на обычные за тем исключением, что *основа* имени файла (часть, не содержащая расширение) представлена символом %. Наш *makefile* выполняет свою работу благодаря трём встроенным правилам. Первое указывает, как произвести компиляцию файла с расширением *.o* из файла с расширением *.c*:

```
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Второе правило указывает, как получить файлы с расширением *.c* из файлов с расширением *.l*:

```
%.c: %.l
    @$(RM) $@
    $(LEX.l) $< > $@
```

Наконец, существует специальное правило получения файла без расширения (он всегда считается исполняемым) из файлов с расширением *.o*:

```
%: %.o
    $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Прежде, чем углубиться в детали синтаксиса, давайте внимательно рассмотрим вывод *make* и разберёмся, как он применяет встроенные правила.

Если запустить *make* с нашим исправленным *makefile*'ом, вывод будет следующим:

```
$ make
gcc  -I include  -c -o count_words.o src/count_words.c
gcc  -I include  -c -o counter.o src/counter.c
flex  -t src/lexer.l > lexer.c
gcc  -I include  -c -o lexer.o lexer.c
gcc   count_words.o counter.o lexer.o /lib/libfl.a -o count_words
rm lexer.c
```

Сначала *make* читает *makefile* и выставляет целью по умолчанию файл *count_words*, поскольку аргументы командной строки не заданы. Рассматривая цель по

умолчанию, *make* определяет её реквизиты: *count_words.o* (этот реквизит не указан в *makefile*, он следует из неявного правила), *counter.o*, *lexer.o* и *-lfl*. Далее *make* пытается собрать каждый из этих реквизитов.

Когда *make* рассматривает первый реквизит, *count_words.o*, он не находит явного правила, однако обнаруживает неявное. Не найдя исходного файла в текущем каталоге, он просматривает директории, указанные в `VPATH`, и обнаруживает исходный файл в каталоге *src*. Поскольку файл *src/count_words.c* не имеет реквизитов, *make* запускает сценарий компиляции *count_words.o*, ассоциированный с неявным правилом. Таким же образом получается файл *counter.o*. Когда *make* рассматривает *lexer.o*, соответствующий исходный файл не обнаруживается, поэтому *make* предполагает, что *lexer.c* является промежуточным файлом, и ищет подходящее правило для его получения. Обнаружив правило создания файла с расширением *.c* из файла с расширением *.l*, *make* замечает, что файл *lexer.l* существует. Поскольку *lexer.l* не имеет реквизитов, вызывается сценарий генерации файла *lexer.c*, содержащий вызов команды *flex*. Затем происходит компиляция с получением объектного файла. Использование последовательностей правил для сборки цели, подобных предыдущим, называется *цепочкой правил*.

Далее, *make* исследует спецификацию библиотеки *-lfl*. Осуществляется поиск в каталогах из списка стандартных каталогов библиотек, результатом которого является файл */lib/libfl.a*.

Теперь *make* имеет все реквизиты для получения *count_words*, запускается команда компоновки *gcc*. Наконец, *make* вспоминает, что был создан промежуточный файл, который больше не нужно хранить, и осуществляет удаление этого файла.

Как мы убедились, использование правил в *makefile*'ах позволяет избежать спецификации многих деталей. Правила могут иметь сложные отношения, которые порождают чрезвычайно широкие возможности. В частности, наличие встроенной базы данных общих правил значительно упрощает спецификацию многих *makefile*'ов.

Встроенные правила могут быть настроены путём изменения значений переменных, фигурирующих в сценариях сборки. Типичные сценарии имеют множество переменных, начиная с имени команды, подлежащей выполнению, и заканчивая большими группами опций командной строки, такими, как имя выходного файла, флаги оптимизации, включение символьной информации и т.д. Вы можете увидеть стандартную базу данных *make* с помощью команды `make -print-database`.

2.4.1 Шаблоны

Символ процента в шаблонном правиле практически эквивалентен специальному символу `*` командного интерпретатора Unix. Он соответствует произвольному числу символов. Знак процента может появиться в любом месте шаблона ровно

один раз. Вот примеры допустимого использования этого символа.

```
%,v
s%.o
wrapper_%
```

Символы, входящие в шаблон и отличные от процента, соответствуют самим себе. Шаблон может содержать префикс, суффикс или и то, и другое. Когда *make* ищет соответствие шаблонному правилу, сначала проверяется соответствие шаблону цели. Шаблон цели должен начинаться с префикса и заканчиваться суффиксом (если они есть). Если соответствие найдено, символы между префиксом и суффиксом становятся основой имени файла. Затем *make* производит поиск реквизитов шаблонного правила, подставляя основу имени файла в шаблон реквизита. Если результирующий файл существует или может быть получен с помощью другого правила, соответствие считается установленным и правило применяется. Основа файла должна содержать хотя бы один символ.

Допускается также иметь шаблоны с одним только символом процента. Наиболее частое использование этого шаблона — сборка исполняемых файлов Unix. Например, вот несколько встроенных шаблонных правил, использующихся *make* для сборки программ:

```
%: %.mod
    $(COMPILE.mod) -o $@ -e $@ $^

%: %.cpp

    $(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%: %.sh
    cat $< >$@
    chmod a+x $@
```

Эти правила будут использованы для сборки программы из исходных файлов Modula, исходных файлов C, обработанных препроцессором, или командных сценариев интерпретатора Bourne shell соответственно. Мы рассмотрим другие неявные правила в разделе «База данных неявных правил».

2.4.2 Статические шаблонные правила

Статические шаблонные правила применяются только к определённому списку целей.

```
$(OBJECTS): %.o: %c
        $(CC) -c $(CFLAGS) $< -o $@
```

Единственная разница между этим правилом и обычным шаблонным правилом — наличие спецификации `$(OBJECTS):`. Эта спецификация ограничивает область действия правила файлами, перечисленными в переменной `$(OBJECTS)`. Каждый объектный файл из этой переменной проверяется на соответствие шаблону `%.c`, извлекается основа его имени, которая затем подставляется в шаблон `%.c`, порождая имя реквизита цели. Если файл из списка не соответствует шаблону цели, *make* выдаст предупреждение.

Используйте статические шаблонные правила везде, где легче перечислить файлы целей явно, нежели определять их по расширению или шаблону.

2.4.3 Суффиксные правила

Суффиксные правила — это первоначальный (и ныне устаревший) способ определения неявных правил. Поскольку другие версии *make* могут не поддерживать синтаксис шаблонов GNU *make*, вы ещё можете встретить такие правила в *makefile*'ах дистрибутивов, предназначенных для широкого распространения, поэтому важно уметь читать и понимать их синтаксис. Итак, несмотря на то, что компиляция GNU *make* для целевой платформы является предпочтительным методом переноса *makefile*'ов, при некоторых обстоятельствах вам, возможно, придётся писать суффиксные правила.

Суффиксные правила выглядят как два написанных слитно суффикса, выступающие в роли правила:

```
.c.o:
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Тот факт, что первым указывается суффикс реквизита, а вторым — суффикс цели, может внести некоторую путаницу. Предыдущему правилу соответствуют в точности те же цели и реквизиты, что и следующему:

```
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Суффиксные правила формируют основу имени файла путём удаления суффикса; имя файла реквизита получается в результате замены суффикса цели суффиксом реквизита. Суффиксное правило распознаётся *make* только в том случае, если оба суффикса находятся в списке известных суффиксов.

Предыдущее суффиксное правило называется также двусуффиксным, поскольку оно содержит два суффикса. Существуют также односуффиксные правила. Такие правила содержат только один суффикс — суффикс реквизита. Эти правила используются для сборки исполняемых файлов Unix, не имеющих расширения:

```
.p:
    $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Предыдущее правило описывает создание исполняемого файла из исходного файла Pascal. Оно полностью эквивалентно следующему правилу:

```
%: %.p
    $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Список известных суффиксов является наиболее необычной частью синтаксиса. Для определения этого списка используется специальная цель `.SUFFIXES`. Вот первая часть стандартного определения `.SUFFIXES`:

```
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l
```

Вы можете добавить собственные суффиксы, добавив правило `.SUFFIXES` в ваш *makefile*:

```
.SUFFIXES: .pdf .fo .html .xml
```

Если вы хотите удалить все известные суффиксы (например, они перекрываются с вашими специальными суффиксами), определите цель `.SUFFIXES` как не имеющую реквизитов:

```
.SUFFIXES:
```

Вы также можете использовать опцию командной строки `--no-builtin-rules` (или `-r`).

Мы не будем использовать суффиксные правила в этой книге, поскольку шаблонные правила GNU *make* более выразительны и óбщны.

2.5 База данных неявных правил

GNU *make* 3.80 содержит примерно 90 встроенных неявных правил (шаблонных и суффиксных), предназначенных для работы с исходными файлами C, C++, Pascal, FORTRAN, ratfor, Modula, Texinfo, TEX (включая инструменты *tangle* и *weave*), Emacs Lisp, RCS и SCCS, а также правила для поддержки программ, предназначенных для работы с этими языками: *cpp*, *as*, *yacc*, *lex*, *tangle*, *weave* и инструменты для работы с *dvi*.

Если вы используете одну из этих программ, то, возможно, вас вполне устроят возможности, предоставляемые встроенными правилами. Если же вы используете некоторые не поддерживаемые языки наподобие Java или XML, вам придётся писать правила самим. Впрочем, обычно для добавления поддержки языка достаточно написать всего несколько простых правил.

Чтобы увидеть встроенные правила *make*, нужно запустить его с опцией `--print-data-base` (или просто `-p`). Вывод составит около тысячи строк текста: после номера версии и текста лицензии *make* выведет на экран определения всех переменных

с описанием их «происхождения». Например, переменные могут быть взяты из окружения, являться стандартными или автоматическими. После описания переменных последуют правила. Текущий формат правил GNU *make* выглядит следующим образом:

```
%: %.C
#  Команды для выполнения (встроенные):
    $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Комментарии к правилам, определённым в *makefile*, будут содержать имя файла и номер строки, в которых эти правила определены:

```
%.html: %.xml
#  Команды для выполнения (из `Makefile', строка 168):
    $(XMLTO) $(XMLTO\_FLAGS) html-nochunks $<
```

2.5.1 Работа с неявными правилами

Встроенные правила применяются каждый раз, когда рассматривается цель, для которой не указан сценарий сборки. Таким образом, использовать встроенные правила очень просто — достаточно не указывать команд при добавлении цели в *makefile*. Это будет сигналом для *make*, призывающим к поиску подходящего правила в базе данных встроенных правил. Обычно это приводит к нужному результату, но в редких случаях ваша среда разработки может породить некоторые проблемы. Предположим, у вас имеется смешанная среда разработки, состоящая из исходных файлов C и Lisp. Если файлы *editor.l* и *editor.c* находятся в одном каталоге (например, один из них является низкоуровневой реализацией, к которой имеет доступ другой), *make* будет считать, что файл с исходным кодом на Lisp на самом деле является файлом *flex* (повторим, *flex* использует файлы с расширением *.l*), а исходный файл C — результат запуска *flex*. Если файл *editor.o* является целью, а *editor.l* модифицировался позднее *editor.c*, *make* попытается обновить исходный файл C выводом команды *flex*, заместив ваш исходный код.

Чтобы избежать этой проблемы, вы можете удалить из базы данных правила, вызывающие *flex*, следующим образом:

```
%.o: %.l
%.c: %.l
```

Шаблонное правило без сценария сборки удаляет правило из базы данных *make*. На практике ситуации, подобные описанной, встречаются крайне редко. Тем не менее, важно помнить, что правила из встроенной базы данных могут взаимодействовать с вашими *makefile*'ами неожиданным для вас образом.

Мы уже рассмотрели несколько примеров того, как *make* выстраивает цепочки правил для сборки цели. Такое поведение может порождать довольно сложные

конструкции. Когда *make* рассматривает варианты сборки цели, происходит поиск неявных правил, шаблону цели которых соответствует имя текущей цели. Для каждого подходящего шаблона цели *make* осуществляет проверку соответствия реквизитов. Таким образом, проверка реквизитов осуществляется *сразу* после нахождения подходящего шаблона цели. Если реквизиты найдены, правило применяется. Для некоторых шаблонов цели может быть несколько возможных видов реквизитов. Например, файлы с расширением *.o* можно получить из файлов с расширением *.c*, *.cc*, *.cpp*, *.p*, *.f*, *.r*, *.s* и *.mod*. Если исходные файлы не обнаружены при переборе всех возможных вариантов, *make* произведёт повторный поиск правил, рассматривая исходные файлы в качестве новых целей. Повторяя этот поиск рекурсивно, *make* выстроит цепь правил, позволяющих произвести сборку цели. Мы уже имели возможность убедиться в этом на примере файла *lexer.o*: *make* смог получить *lexer.o* из *lexer.l* даже при отсутствующем файле *lexer.c*, вызвав сначала правило $.l \rightarrow .c$, а затем правило $.c \rightarrow .o$.

Давайте рассмотрим одну из самых впечатляющих цепочек, которую *make* может породить с помощью встроенной базы данных правил. Сначала создадим пустой исходный файл *yacc* и зарегистрируем его в системе контроля ревизий RCS командой *ci*:

```
$ touch foo.y
$ ci foo.y
foo.y,v  <--  foo.y
.
initial revision: 1.1
done
```

Теперь попросим *make* создать исполняемый файл *foo*. Опция `--just-print` (или просто `-n`) означает, что от *make* требуется лишь описать, какие действия будут выполнены. Заметим, что у нас нет *makefile*'а и исходного кода, только RCS-файл.

```
$ make -n foo
co  foo.y,v foo.y
foo.y,v  -->  foo.y
revision 1.1
done
bison -y  foo.y
mv -f y.tab.c foo.c
gcc -c -o foo.o foo.c
gcc foo.o -o foo
rm foo.c foo.o foo.y
```

Следуя по цепочке неявных правил и реквизитов, *make* определяет, что сборка исполняемого файла *foo* возможна при наличии объектного файла *foo.o*, который

можно получить из исходного файла *foo.c*. В свою очередь, *foo.c* может быть получен из файла *yacc foo.y*, доступного при извлечении его из файла RCS *foo.y,v*, имеющегося в наличии. Составив план действий, *make* выполняет извлечение *foo.y* с помощью команды *co*, преобразует его в исходный файл *foo.c* командой *bison*, компилирует полученный исходный файл C в объектный файл *foo.o* вызовом *gcc* и производит компоновку для получения исполняемого файла *foo* повторным вызовом *gcc*. Всё это происходит с использованием лишь встроенной базы данных правил. Впечатляет.

Файлы, порождаемые применением цепочки правил, называются *промежуточными* и обрабатываются *make* особым образом. Во-первых, поскольку промежуточные файлы не специфицируются в качестве целей (иначе они бы не были промежуточными), *make* никогда не удовлетвориться просто сборкой этого файла. Во-вторых, поскольку *make* создаёт эти файлы самостоятельно как побочные эффекты сборки цели, такие файлы должны быть удалены перед завершением работы. Вы можете убедиться в этом, посмотрев на последнюю строку предыдущего примера.

2.5.2 Структура правил

Встроенные правила имеют стандартную структуру, направленную на предоставление возможности простой настройки этих правил. Рассмотрим сначала общую структуру, затем обсудим тонкости настройки. Ниже представлено уже знакомое правило компиляции исходного файла C в объектный файл:

```
%.o: %.c
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Настройка этого правила осуществляется за счёт переменных, используемых им. Мы видим две переменные, однако переменная `$(COMPILE.c)`, например, определена через другие переменные:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET\_ARCH) -c
CC = gcc
OUTPUT_OPTION = -o $@
```

Таким образом, замена компилятора языка C может быть произведена изменением значения переменной `CC`. Другие переменные используются для определения опций компиляции (`CFLAGS`), опций препроцессора (`CPPFLAGS`) и архитектурно-зависимых опций (`TARGET_ARCH`).

Все переменные, использующиеся во встроенных правилах, нацелены на максимально простую настройку правил. По этой причине очень важно быть осторожным при определении значений этих переменных в вашем *makefile*'е. Если вы будете определять переменные «по наитию», вы можете нарушить возможность

настройки правил конечным пользователем. Рассмотрим пример присваивания в *makefile*'е:

```
CPPFLAGS = -I project/include
```

Если пользователь хочет самостоятельно определить значение переменной в командной строке, он обычно поступает следующим образом:

```
$ make CPPFLAGS=-DDEBUG
```

Однако такой вызов случайно удалит опцию `-I` (которая, предположительно, необходима для компиляции) поскольку значения переменных, определённые в командной строке, перекрывают все другие присваивания этим переменным[2]. Таким образом, ненадлежащее определение переменной `CPPFLAGS` нарушило возможность настройки, на наличие которой рассчитывает бóльшая часть пользователей. Вместо использования простых присваиваний, рассмотрим переопределение переменной компиляции с добавлением новых переменных:

```
COMPILE.c = $(CC) $(CFLAGS) $(INCLUDES) $(CPPFLAGS) $(TARGET_ARCH) -c
INCLUDES = -I project/include
```

Ещё одним выходом из ситуации является использование доопределения переменных вместо присваивания, этот подход обсуждается в разделе «Другие виды присваивания» главы 3.

2.5.3 Неявные правила для управления ревизиями

В *make* реализована на уровне встроенных правил поддержка двух систем контроля ревизий: RCS и SCCS. Складывается впечатление, что искусство управления исходным кодом на текущем этапе своего развития и современная компьютерная инженерия оставили *make* далеко позади. На практике поддержка управления ревизиями в *make* практически не используется. И на это есть ряд причин.

Во-первых, инструменты контроля ревизий, поддерживаемые *make*, RCS и SCCS, в прошлом весьма значимые и ценные, были повсеместно вытеснены CVS (Concurrent Version System) или коммерческими инструментами. Несмотря на то, что CVS использует RCS для внутреннего управления одиночными файлами, прямое использование RCS порождает значительные проблемы, когда проект состоит из более чем одного каталога, или в нём задействовано более одного разработчика. В частности, CVS была разработана для заполнения пробелов в функциональности RCS

[2] Для более подробных сведений о присваиваниях в командной строке следует обратиться к разделу «Где определяются переменные».

в упомянутых аспектах. Поддержки CVS в *make* никогда не было, и это, возможно, является правильным решением[3].

Общепризнано, что жизненный цикл разработки программного обеспечения становится всё более сложным. Приложения редко плавно продвигаются от одного релиза к другому. Как правило, одновременно может использоваться (и, следовательно, требовать исправления ошибок) несколько версий приложения, в то время как ещё несколько версий могут находиться в активной разработке. CVS предоставляет богатые возможности управления параллельной разработкой программного обеспечения. Однако это требует от разработчика чёткого осознания того, над какой версией исходного кода он работает. Если бы *make* автоматически извлекал из хранилища исходный код для компиляции, сразу бы вставали вопросы актуальности извлечённой версии и совместимости этой версии с кодом, расположенным в рабочих каталогах разработчика. Во многих проектах разработчики работают с тремя и более различными версиями программного продукта в течение одного дня. Проверка целостности сложной системы достаточно сложна и без наличия инструмента, безмолвно изменяющего ваш исходный код.

К тому же, одной из самых важных возможностей CVS является возможность доступа к удалённому хранилищу. В большинстве проектов CVS хранилище (база данных версионных файлов) располагается на сервере, а не на машине разработчика. Несмотря на то, что удалённый доступ достаточно быстр (по крайней мере, в локальных сетях), запуск *make*, проверяющего каждый файл на удалённом сервере, не является хорошей идеей, поскольку производительность упадёт катастрофически.

Таким образом, можно использовать встроенные правила для довольно прозрачного взаимодействия с RCS и SCCS, но правил для доступа к хранилищам CVS для поиска файлов не существует. По большому счёту, даже если бы такие правила существовали, им трудно было бы найти разумное применение. С другой стороны, использование CVS для управления версиями *makefile*'ов чрезвычайно полезно и порождает много интересных применений, таких как проверка правильности помещения файла в хранилище, управление нумерацией релизов и корректное выполнение автоматического тестирования. Все это возможно при использовании CVS авторами *makefile*'ов, а не за счёт интеграции *make* с CVS.

2.5.4 Пример простой справки

Большие *makefile*'ы могут содержать много целей, трудных для запоминания. Одним из способов устранения этой проблемы является выбор в качестве цели по умолчанию абстрактной цели вывода краткой справки. Однако поддержка текста

[3]CVS, в свою очередь, постепенно вытесняется более современными инструментами. На данный момент наиболее часто встречается система управления исходным кодом Subversion (*http://subversion.tigris.org*) (прим. автора).

этой справки в актуальном состоянии является довольно утомительным занятием. К счастью, для составления справки могут быть использованы команды из встроенной базы данных правил *make*. Ниже приведён пример цели, выводящей отсортированный список доступных целей:

```
# help - цель по умолчанию
.PHONY: help
help:
    $(MAKE) --print-data-base --question |            \
    $(AWK) '/\^[\^.\%][-A-Za-z0-9\_]*:/               \
           { print substr($$1, 1, length($$1)-1) }' | \
    $(SORT) |                                         \
    $(PR) --omit-pagination --width=80 --columns=4
```

Сценарий состоит из одного конвейера программ. Список правил *make* выводится при помощи ключа `--print-data-base`, ключ `--question` исключает выполнение сценариев сборки. Затем база данных пропускается через простой сценарий *awk*, извлекающий из потока правил все цели, имена которых не начинаются с символов процента или точки (то все шаблонные и суффиксные правила, соответственно) и вырезающий всю прочую информацию в строке. Наконец, цели сортируются по имени и выводятся в четыре колонки.

Другим возможным решением проблемы является применение специального сценария *awk* непосредственно к *makefile*'у. Это потребует специальной обработки включения *makefile*'ов (см. раздел «Директива `include`» главы 3) и сделает невозможным обработку правил, созданных самим *make* на основе шаблонных и суффиксных правил. Версия, представленная выше, избавлена от этих недостатков благодаря вызову *make*, осуществляющему все необходимые действия самостоятельно.

2.6 Специальные цели

Специальной целью называют встроенную абстрактную цель, предназначенную для спецификации поведения *make*. Например, уже известная нам `.PHONY` является специальной целю, реквизиты которой не связаны с реальными файлами и всегда требуют обновления.

Для специальных целей используется стандартный синтаксис *`цель: реквизиты`*, но *`цель`* не является файлом или обычной абстрактной целью. Более всего специальные цели похожи на директивы, изменяющие внутренние алгоритмы *make*.

Существует двенадцать специальных целей. Их можно разделить на три категории: одни изменяют поведение *make*, вторые являются просто флагами, наконец, специальная цель `.SUFFIXES` используется для спецификации суффиксных правил (обсуждавшихся в разделе «Суффиксные правила»).

Наиболее полезны следующие специальные цели:

`.INTERMEDIATE`
: Реквизиты этой цели интерпретируются как промежуточные файлы. Если *make* создаст файл во время сборки другой цели, файл будет автоматически удалён перед завершением работы *make*. Если файл уже существует в момент сборки цели, файл не будет удалён.

  Такое поведение может быть очень полезным при построении цепочки правил. Например, большинство Java утилит принимают списки файлов в стиле Windows. Создание правил для сохранения списков файлов и спецификация их как промежуточных позволяет *make* автоматически удалять множество временных файлов.

`.SECONDARY`
: Реквизиты этой специальной цели интерпретируются как промежуточные файлы, которые не удаляются автоматически. Наиболее часто `.SECONDARY` используется для пометки объектных файлов, хранимых в виде библиотек. Обычно такие объекты будут удалены сразу после добавления их в архив. Иногда удобнее не удалять объектные файлы во время разработки.

`.PRECIOUS`
: Когда *make* аварийно завершает выполнение, файл цели может быть удалён, если он изменился с момента старта *make*. Таким образом избегается возможность сохранения частично собранных (и, возможно, повреждённых) файлов. Иногда вы можете захотеть от *make* другого поведения, например, если файл велик и требует много вычислений для своего создания. Если вы укажете имя такого файла в качестве реквизитов цели `.PRECIOUS`, *make* не будет удалять этот файл в случае аварийного завершения. Эта цель используется довольно редко, но если её применение действительно необходимо, её наличие сохранит разработчику много времени и сил. Заметим, что *make* не осуществляет автоматического удаления в случае ошибки в выполняемой команде, только в случае получения сигнала останова.

`.DELETE_ON_ERROR`
: Эта цель — противоположность `.PRECIOUS`. Спецификация файла как реквизита этой цели означает, что файл должен быть удалён в случае любой ошибки в сценарии сборки, ассоциированном с соответствующим правилом. Обычно *make* удаляет цель только в случае получении сигнала останова.

Остальные специальные цели будут рассмотрены в момент их непосредственного использования. Цели, относящиеся к параллельному выполнению, будут рассмотрены в главе 10, цель `.EXPORT_ALL_VARIABLES` — в главе 3.

2.7 Автоматическое определение зависимостей

Когда мы изменили нашу программу подсчёта слов так, чтобы часть объявлений была описана в заголовочных файлах, мы, сами того не замечая, добавили новую проблему. Мы описали зависимости между объектными и заголовочными файлами в наш *makefile* самостоятельно. В нашем случае сделать это было нетрудно, но в реальных программах (а не в игрушечных примерах) это может быть весьма утомительным и порождающим ошибки процессом. На самом деле, в большинстве программ указание зависимостей практически невозможно, поскольку заголовочные файлы могут включать другие заголовочные файлы, образуя сложное дерево включений. Например, в моей системе один заголовочный файл *stdio.h* (наиболее часто используемый заголовочный файл стандартной библиотеки языка C) в общем счёте включает 15 других заголовочных файлов. Разрешение подобных зависимостей вручную является практически безнадёжным занятием. Однако неудавшаяся компиляция ведёт к часам потраченного на отладку времени или, что ещё хуже, к проблемам в уже выпущенном программном обеспечении. Что же нам делать?

К счастью, компьютеры весьма хорошо справляются с задачами поиска и нахождения соответствий шаблону. Давайте используем программу для определения зависимостей между исходными файлами, и даже записи этих зависимостей в соответствии со стандартным синтаксисом *make*. Как вы, возможно, уже догадались, такая программа уже существует, по крайней мере, для исходных файлов на C/C++. Компилятор *gcc*, как и многие другие компиляторы, имеет опцию для чтения исходных файлов и составления зависимостей для *make*. Например, так мы можем определить зависимости для *stdio.h*

```
$ echo "#include <stdio.h>" > stdio.c
$ gcc -M stdio.c
stdio.o: stdio.c /usr/include/stdio.h /usr/include/_ansi.h \
/usr/include/newlib.h /usr/include/sys/config.h \
/usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stddef.h \
/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/stdarg.h \
/usr/include/sys/reent.h /usr/include/sys/_types.h \
/usr/include/sys/types.h /usr/include/machine/types.h \
/usr/include/sys/features.h /usr/include/cygwin/types.h \
/usr/include/sys/sysmacros.h /usr/include/stdint.h \
/usr/include/sys/stdio.h
```

«Отлично,» — скажете вы, —«Теперь мне придётся запускать gcc, открывать текстовый редактор и вставлять результаты работы компилятора с ключом `-M` в свой *makefile*. Какой ужас.». И вы были бы правы, если бы это была вся правда. Существует два стандартных способа включения автоматически составленных за-

висимостей в *makefile*. Первый, он же самый старый, заключается в добавлении комментария наподобие следующего:

```
# Далее следуют автоматически составленные зависимости:
# НЕ РЕДАКТИРОВАТЬ
```

в конец *makefile*'а и написании сценария командного интерпретатора для автоматического обновления этого раздела. Это, безусловно, гораздо лучше ручного обновления, но всё ещё довольно неудобно. Второй метод заключается в добавлении директивы include. Бóльшая часть версий *make* поддерживает эту директиву, и, безусловно, GNU *make* в их числе. Идея заключается в спецификации цели, с которой ассоциированы действия по запуску *gcc* с ключом `-M`, сохранении результатов в файле зависимостей и повторный запуск *make* с включением составленного файла зависимостей в основной *makefile*. До появления GNU *make* это делалось правилом следующего вида:

```
depend: count\_words.c lexer.c counter.c
    $(CC) -M $(CPPFLAGS) $^ > $@
include depend
```

Сначала вы запускаете *make* с целью составить файл зависимостей, и только после этого производите повторный пуск для сборки программы. На момент появления этой возможности она выглядела неплохо, однако часто люди добавляли или удаляли зависимости из исходного кода, забыв заново составить файл зависимостей. Это становилось причиной неправильной компиляции со всеми вытекающими неприятностями. GNU *make* решил эту неприятную проблему с помощью мощной функциональности и довольно простого алгоритма. Рассмотрим сначала алгоритм. Если мы составим для каждого исходного файла собственный файл зависимостей, скажем, файл с расширением *.d*, и добавим этот файл в качестве цели к соответствующему правилу, то сможем сообщить *make*, что *.d* файл нуждается в обновлении (наряду с объектным файлом) при изменении исходного файла:

```
counter.o counter.d: src/counter.c include/counter.h include/lexer.h
```

Составление этого правила может быть завершено шаблонным правилом и довольно неуклюжим сценарием (взятым прямо из руководства по GNU *make*) [4]:

[4] Этот довольно выразительный сценарий, по моему мнению, всё же требует некоторого объяснения. Сначала мы используем компилятор C с опцией `-M` для создания временного файла, содержащего список зависимостей цели. Имя временного файла получается из названия цели `$@` и добавочного уникального числового суффикса `.$$$$`. В командном интерпретаторе *sh* переменная `$$` содержит идентификатор текущего запущенного процесса командного интерпретатора. Поскольку этот идентификатор является уникальным, имя нашего временного файла также получается уникальным. Затем мы используем *sed* для добавления файла с расширением *.d* в

```
%.d: %.c
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                  \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

Теперь рассмотрим вышеупомянутую функциональность. GNU *make* будет рассматривать каждый включаемый файл в качестве цели, нуждающейся в обновлении. Таким образом, когда мы будем упоминать *.d* файлы, *make* автоматически попытается создать эти файлы во время чтения *makefile*'а. Ниже представлен наш пример с добавлением автоматического управления зависимостями:

```
VPATH    = src include
CPPFLAGS = -I include
SOURCES  = count_words.c \
           counter.c     \
           lexer.c
count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h

include $(subst .c,.d,$(SOURCES))

%.d: %.c
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                  \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

Директива включения должна появляться только после записанных вручную правил, чтобы не подменить цель по умолчанию целью из включаемого файла. Директива `include` принимает в качестве аргумента список файлов (чьи имена могут включать шаблоны). В предыдущем примере мы использовали встроенную функцию *make substr* для трансформации списка исходных файлов в список файлов зависимостей (мы рассмотрим *substr* более подробно в разделе «Строковые функции» главы 4). Пока просто примите к сведению, что мы используем эту функцию для замены строки *.c* на строку *.d* в каждом слове списка `$(SOURCES)`.

Если теперь мы запустим *make* с опцией `--just-print`, то получим следующее:

```
$ make --just-print
```

качестве цели правила. Выражение *sed* состоит из шаблона поиска `\($\)\1.o[ :]*` и подстановки `\1.o $@ :`, разделённых запятыми. Шаблон поиска состоит из основы имени цели `$*`, заключенной в группу регулярного выражения `\(\)`, за которой следует суффикс `.o`. После имени цели могут следовать пробелы или двоеточия (`[ :]*`). Подстановка восстанавливает первоначальную цель с помощью ссылки на первую группу регулярного выражения с добавлением суффикса (`\1.o`) и добавляет файл зависимостей в качестве второй цели правила (`$@`).

```
Makefile:13: count_words.d: No such file or directory
Makefile:13: lexer.d: No such file or directory
Makefile:13: counter.d: No such file or directory
gcc -M -I include src/counter.c > counter.d.$$;       \
sed 's,\(counter\)\.o[ :]*,\1.o counter.d : ,g'       \
< counter.d.$$ > counter.d;                           \
rm -f counter.d.$$
flex -t src/lexer.l > lexer.c
gcc -M -I include lexer.c > lexer.d.$$;           \
sed 's,\(lexer\)\.o[ :]*,\1.o lexer.d : ,g'       \
< lexer.d.$$ > lexer.d;                           \
rm -f lexer.d.$$
gcc -M -I include src/count_words.c > count_words.d.$$; \
sed 's,\(count_words\)\.o[ :]*,\1.o count_words.d : ,g' \
< count_words.d.$$ count_words.d;                       \
rm -f count_words.d.$$
rm lexer.c
gcc -I include -c -o count_words.o src/count_words.c
gcc -I include -c -o counter.o src/counter.c
gcc -I include -c -o lexer.o lexer.c
gcc count_words.o counter.o lexer.o /lib/libfl.a -o count_words
```

Сначала *make* выводит несколько предупреждений, с виду напоминающих ошибки. Не стоит волноваться, это всего лишь предупреждения. *make* производит поиск файлов, указанных в директиве `include`, не находит их, и перед началом поиска правила для создания этих файлов выводит предупреждение `No such file or directory`. Эти предупреждения могут быть подавлены при помощи символа `-`, добавленного перед директивой `include`. Следующие строки демонстрируют вызов *gcc* с опцией `-M` и запуск команды *sed*. Обратите внимание на то, что *make* должен вызвать *flex* для создания *lexer.c*, удаляемый перед началом сборки цели по умолчанию.

Теперь у вас есть представление об автоматическом определении зависимостей. Эта тема содержит ещё много интересных вопросов, например, построение зависимостей для других языков программирования, или вывод зависимостей в виде дерева. Мы вернёмся к этим темам во второй части книги.

2.8 Управление библиотеками

Библиотечный архив (*archive library*), обычно называемый просто библиотекой или архивом, — это специальный файл, содержащий в себе другие файлы, именуемые *элементами архива* (*archive members*). Например, стандартная библиотека языка C *libc.a* содержит низкоуровневые функции. Библиотеки используются настолько часто, что *make* имеет специализированную функциональность для создания, поддержки и компоновки архивов. Архивы создаются и модифицируются при помощи программы *ar*.

Давайте вернёмся к нашему примеру. Мы можем модифицировать нашу программу подсчёта слов, упаковав все её компоненты, пригодные для повторного использования, в библиотеку. Наша библиотека будет состоять из двух файлов: *counter.o* и *lexer.o*. Для создания библиотеки вызовем команду *ar*:

```
$ ar rv libcounter.a counter.o lexer.o
a - counter.o
a - lexer.
```

Опции `rv` означают, что мы хотим заменить элементы библиотеки указанными объектными файлами, и что *ar* должен выводить отчёт о своих действиях. Мы можем использовать действие замены даже в том случае, если указанная библиотека не существует. Первым аргументом после опций является имя библиотеки, за ним следуют имена объектных файлов (некоторые версии *ar* требуют опции *c* в случае, если библиотека ещё не существует, но GNU *ar* не требует этого). Две строки, следующие за вызовом команды *ar*, являются отчётом о том, что объектные файлы были добавлены в библиотеку.

Использование опции замены позволяет создавать и изменять архив последовательно:

```
$ ar rv libcounter.a counter.o
r - counter.o
$ ar rv libcounter.a lexer.o
r - lexer.o
```

Теперь *ar* предваряет имена файлов символом "r". Это значит, что файлы в архиве были заменены.

Библиотека может быть скомпонована в исполняемый файл несколькими способами. Самый простой способ — просто указать имя библиотеки в списке аргументов компилятора. В свою очередь, компилятор или компоновщик будут использовать расширение для определения типа каждого из указанных в командной строке файлов:

```
cc count_words.o libcounter.a /lib/libfl.a -o count_words
```

Компилятор *cc* распознает два файла *libcounter.a* и */lib/libfl.a* как библиотеки и будет искать в них недостающие символы. Ещё одним способом ссылки на библиотеку является опция `-l`:

```
cc count_words.o -lcounter -lfl -o count_words
```

Как вы можете видеть, при использовании этой опции опускается префикс и суффикс имени библиотеки. Опция `-l` делает командную строку более компактной и удобочитаемой, однако, при использовании этой опции вы получаете гораздо

более весомое преимущество. Когда компилятор *cc* видит опцию `-l`, он *ищет* библиотеку в стандартных каталогах системных библиотек. Это избавляет программиста от необходимости знать точный путь к файлу библиотеки и делает команду компоновки более переносимой. К тому же, в системах, поддерживающих разделяемые библиотеки (библиотеки с расширением *.so* на системах семейства Unix), компоновщик будет искать сначала разделяемые библиотеки, и только если подходящей не обнаружено, будет осуществлён поиск библиотечного архива. Такой подход позволяет программам пользоваться преимуществами разделяемых библиотек без их явной спецификации. Таково стандартное поведение компилятора и компоновщика GNU. Старые компоновщики и компиляторы могут не осуществлять такой оптимизации.

Список каталогов, в которых компилятор должен осуществлять поиск библиотек, может быть изменён с помощью опции `-L`, указывающей список и порядок каталогов, в которых нужно искать библиотеки. Эти каталоги будут добавлены в список прямо перед системными каталогами библиотеки и будут использоваться для всех опций `-l` в командной строке. На самом деле, компиляции в предыдущем примере не завершится успехом, поскольку текущий каталог не входит в список каталогов библиотек *cc*. Мы можем решить эту проблему добавлением текущего каталога в список как показано ниже:

```
cc count_words.o -L. -lcounter -lfl -o count_words
```

Библиотеки вносят некоторые трудности в процесс сборки программ. Какие возможности предоставляет *make* для упрощения этого процесса? GNU *make* включает функциональность как по созданию библиотек, так и использованию библиотек при компоновке. Давайте посмотрим, как это работает.

2.8.1 Создаём и изменяем библиотеки

Библиотеки фигурируют в *makefile*'е в качестве обычных файлов. Ниже представлено простое правило для создания нашей библиотеки:

```
libcounter.a: counter.o lexer.o
    $(AR) $(ARFLAGS) $@ $^
```

Это правило использует встроенные переменные `AR` и `ARFLAGS`, содержащие имя программы *ar* и стандартные опции `rv` соответственно. Для спецификации файла архива используется автоматическая переменная `$@`, а для спецификации реквизитов — автоматическая переменная `$^`.

Теперь, если вы укажите файл *libcounter.a* в качестве реквизита цели `count_words`, *make* обновит нашу библиотеку перед компоновкой исполняемого файла. Обратите внимание на одну деталь. *Все* элементы архива будут замещены, даже

если среди них есть не изменявшиеся с момента последнего обновления архива элементы. Чтобы не терять время впустую, мы можем написать более подходящее правило:

```
libcounter.a: counter.o lexer.o
    $(AR) $(ARFLAGS) $@ $?
```

Если вы используете `$?` вместо `$^`, *make* будет подставлять в список аргументов только те объектные файлы, которые имеют более позднюю дату модификации, чем цель.

Можем ли мы ещё улучшить это правило? Может быть да, а может и нет. *make* имеет встроенную поддержку обновления отдельных файлов в архиве, но прежде, чем мы вдадимся в эти детали, стоит сделать несколько важных замечаний относительно такого подхода к работе с библиотеками. Одна из основных задач *make* состоит в том, чтобы эффективно использовать время процессора и собирать только те файлы, которые действительно в этом нуждаются. К сожалению, вызов *ar* для каждого элемента архива по отдельности при наличии несколько десятков файлов занимает настолько много времени, что перевешивает преимущество элегантного синтаксиса, рассмотренного далее. Используя простой метод, представленный выше, мы можем вызвать *ar* один раз для всех изменившихся файлов и избежать множества ненужных системных вызовов *fork/exec*. Кроме того, на многих системах использование ключа `r` при вызове *ar* очень неэффективно. На моём компьютере 1.9 GHz Pentium 4 создание большого архива, содержащего 14216 элементов общим размером 55 MB, занимает 4 минуты 24 секунды, в то время как замена одного элемента в этом архиве требует 28 секунд. Таким образом, создание архива заново будет более быстрой альтернативой замене элементов при наличии более 10 (из 14216!) изменившихся файлов. В такой ситуации более разумным подходом будет единовременное обновление архива с использованием автоматической переменной `$?`. Для небольших библиотек и более быстрых компьютеров нет причин отказываться от элегантного подхода, описанного ниже, в пользу более простого, но и более быстрого.

В GNU *make* элемент архива может быть специфицирован при помощи следующей нотации:

```
libgraphics.a(bitblt.o): bitblt.o
    $(AR) $(ARFLAGS) $@ $<
```

Здесь *libgraphics.a* — это имя библиотеки, а *bitblt.o* (сокращение от *bit block transfer, передача битовых блоков*) — имя её элемента. Синтаксис *libgraphics.a(bitblt.o)* означает модуль, содержащийся в библиотеке. Реквизитом для цели является сам объектный файл, а командой — добавление этого файла в архив. Автоматическая переменная `$<` используется для получения первого реквизита. На самом деле существует встроенное шаблонное правило, предоставляющее в точности ту же функциональность.

Когда мы соединим всё это воедино, наш *makefile* будет выглядеть следующим образом:

```
VPATH    = src include
CPPFLAGS = -I include

count_words: libcounter.a /lib/libfl.a

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

libcounter.a(lexer.o): lexer.o
    $(AR) $(ARFLAGS) $@ $<

libcounter.a(counter.o): counter.o
    $(AR) $(ARFLAGS) $@ $<

count_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h
```

При запуске *make* выводит следующее:

```
$ make
gcc -I include -c -o count_words.o src/count_words.c
flex -t src/lexer.l > lexer.c
gcc -I include -c -o lexer.o lexer.c
ar rv libcounter.a lexer.o
ar: creating libcounter.a
a - lexer.o
gcc -I include -c -o counter.o src/counter.c
ar rv libcounter.a counter.o
a - counter.o
gcc count_words.o libcounter.a /lib/libfl.a -o count_words
rm lexer.c
```

Обратите внимание на правило обновления архива. Автоматическая переменная `$@` приняла значение имени библиотеки, несмотря на то, что имя цели в *makefile*'е было *libcounter.a(lexer.o)*.

Наконец, нужно отметить, что библиотечный архив включает индекс всех символов, содержащихся в нём. Новые программы архиваторов, такие как GNU *ar*, обновляют этот индекс автоматически при добавлении в архив нового символа. Более старые версии архиваторов могут этого не делать. Для создания и обновления индекса архива используется программа *ranlib*. В системах со старой версией архиватора должно использоваться правило следующего вида:

```
libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)
    $(RANLIB) $@
```

Вы также можете использовать альтернативный подход для больших архивов:

```
libcounter.a: counter.o lexer.o
    $(RM) $@
    $(AR) $(ARFLGS) $@ $^
    $(RANLIB) $@
```

Конечно, синтаксис управления элементами архива может использоваться с применением встроенных правил. GNU *make* содержит встроенные правила обновления архивов. Если мы используем эти правила, наш *makefile* будет выглядеть следующим образом:

```
VPATH    = src include
CPPFLAGS = -I include

count_words: libcounter.a -lfl

libcounter.a: libcounter.a(lexer.o) libcounter.a(counter.o)

count_words.o: counter.h

counter.o: counter.h lexer.h

lexer.o: lexer.h
```

2.8.2 Использование библиотек в качестве реквизитов

Когда библиотеки появляются в качестве реквизитов, они могут быть обозначены с помощью расширения файла или опции `-l`. Если указать имя файла библиотеки:

```
xpong: $(OBJECTS) /lib/X11/libX11.a /lib/X11/libXaw.a
    $(LINK) $^ -o $@
```

то компоновщик просто прочитает библиотечные файлы из командой строки. При использовании опции `-l` реквизиты вовсе не выглядят обычными файлами:

```
xpong: $(OBJECTS) -lX11 -lXaw
    $(LINK) $^ -o $@
```

Когда в реквизитах используется форма `-l`, *make* производит поиск библиотеки (предпочитая разделяемую версию) и подставляет абсолютный путь в переменные `$^` и `$?`. Одно из преимуществ такого подхода состоит в возможности производить автоматический поиск библиотек даже в том случае, если компоновщик в вашей системе не поддерживает такой возможности. Другим преимуществом является возможность настройки путей поиска *make*, что позволяет вам производить поиск

собственных библиотек наравне с системными. В приведённом примере первая форма (с использованием абсолютных путей) будет игнорировать разделяемые библиотеки. При использовании же второй формы *make* будет знать, что разделяемые библиотеки более предпочтительны, поэтому сначала произведёт поиск разделяемой версии *X11*, и только в случае неудачи будет выбрана статическая библиотека. Шаблоны для распознавания имён библиотек хранятся в виде реквизитов специальной цели `.LIBPATTERNS` и могут быть настроены для различных форматов имён библиотек.

К сожалению, есть одна неприятная мелочь. Если в какая-либо цель в *makefile*'е специфицирует библиотеку, на неё нельзя ссылаться в реквизитах с помощью опции `-l`. Например, запуск *make* с таким *makefile*'ом:

```
count_words: count_words.o -lcounter -lfl
    $(CC) $^ -o $@ libcounter.a: libcounter.a(lexer.o)

libcounter.a(counter.o)
```

завершится неудачей со следующей ошибкой:

```
No rule to make target ‘-lcounter’, needed by ‘count_words’
```

Причиной ошибки является то, что *make* не совершил подстановку *libcounter.a* вместо *-lcounter* и поиск подходящей цели. Вместо этого был осуществлён обычный поиск библиотеки. Таким образом, для библиотек, собранных в *make*, должно указывается непосредственно имя файла.

Компоновка больших программ без возникновения ошибок подобна искусству чёрной магии. Компоновщик производит поиск библиотек в том порядке, в каком они указаны в командной строке. Таким образом, если библиотека A содержит неопределённый символ, например, *open*, определённый в библиотеке B, то A должна быть указана в командной строке *перед* B (именно так, A требует B). Иначе, когда компоновщик прочитает A и не найдёт определения символа *open*, будет слишком поздно возвращаться назад к B. Компоновщик никогда не осуществляет поиск в уже просмотренных библиотеках. Таким образом, порядок появления библиотек в командной строке играет фундаментальное значение.

Когда реквизиты цели сохраняются в переменных `$^` и `$?`, порядок их следования также сохраняется. Это справедливо даже для реквизитов, размещённых в нескольких правилах. В этом случае реквизиты каждого правила добавляются к списку реквизитов в том порядке, в котором они появляются.

Родственной проблемой является проблема перекрёстных ссылок между библиотеками,также известных как *циклические ссылки* (*circular references*) или *зацикливания* (*circularities*). Предположим, что после некоторой модификации библиотека B использует символ из A. Мы уже знаем, что A должна быть указана до B, но теперь ещё и B должно быть указана до A. Решением является ссылка

на A и до, и после ссылки на B: `-lA -lB -lA`. В больших и сложных программах библиотеки часто должны повторяться подобным образом, иногда более одного раза.

Такая ситуация ставит небольшую проблему при использовании *make*, поскольку автоматические переменные, как правило, не содержат дубликатов. Например, предположим, что нам нужно повторить библиотеку в реквизитах для устранения циклических ссылок:

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
    $(CC) $^ -o $@
```

Этот список реквизитов после подстановки переменных будет выглядеть следующим образом:

```
gcc xpong.o libui.a libdynamics.a /usr/lib/X11R6/libX11.a -o xpong
```

Для подавления последствий такого поведения переменной `$^` в *make* была добавлена переменная `$+`. Эта переменная идентична `$^` с той лишь разницей, что в списке реквизитов сохраняются дубликаты. Используем `$+`:

```
xpong: xpong.o libui.a libdynamics.a libui.a -lX11
    $(CC) $+ -o $@
```

Теперь список реквизитов породит следующую команду компоновки:

```
gcc xpong.o libui.a libdynamics.a libui.a \
/usr/lib/X11R6/libX11.a -o xpong
```

2.8.3 Правила с двойным двоеточием

Правила с двойным двоеточием — это реализация функциональности, позволяющей собирать одну и ту же цель с помощью разных сценариев, в зависимости от того, какое из подмножеств реквизитов было модифицировано. Обычно если цель появляется более одного раза, все её реквизиты соединяются в один список, сценарий сборки же для одной цели может быть указан только один раз. При использовании же правил с двойным двоеточием каждое появление цели рассматривается как отдельное правило и обрабатывается индивидуально. Это значит, что для какой-то определённой цели все правила должны быть одного типа: либо с одним двоеточием, либо с двумя.

По-настоящему полезные применения этой возможности придумать довольно сложно, поэтому давайте рассмотрим следующий искусственный пример:

```
file-list:: generate-list-script
    chmod +x $<
    generate-list-script $(files) > file-list

file-list:: $(files)
    generate-list-script $(files) > file-list
```

Мы можем создать цель `file-list` двумя способами. Если сценарий составления списка файлов изменился, то добавим файлу сценария права на запуск и выполним его. Если изменились исходные файлы, мы просто запускаем сценарий. Несмотря на свою надуманность, пример наглядно демонстрирует, как можно использовать эту функциональность.

Мы рассмотрели большую часть функциональности *make*, связанной с правилами, которые, наряду с переменными и сценариями, составляют самую сущность *make*. Мы фокусировали внимание главным образом на специфику синтаксиса и поведение различных возможностей, практически не останавливаясь на способах их применения в более сложных ситуациях. Это будет главным объектом нашего внимания во второй части книги. А сейчас продолжим обсуждение переменных и команд.

Глава 3

Переменные и макросы

Мы уже видели переменные в *makefile*'ах и множество примеров их использования как во встроенных, так и в определённых пользователем правилах. Однако те примеры, которые мы видели, являются лишь вершиной айсберга. Переменные и макросы могут быть гораздо более сложными. Именно они придают GNU *make* часть его невероятной мощи.

Прежде, чем мы продолжим, важно осознать, что *make* является смешением двух языков. Первый язык описывает граф зависимостей, состоящий из целей и реквизитов (этот язык был подробно рассмотрен в главе 2). Второй язык является макроязыком для осуществления текстовых подстановок. Быть может, вы знакомы и с другими макроязыками: препроцессор C, *m4*, TEX и макроассемблеры. Как и эти макроязыки, *make* позволяет определять условные обозначения для длинной последовательности символов и использовать их в вашей программе. Макропроцессор распознает их в тексте программы и заменит на соответствующую последовательность. Несмотря на то, что удобно думать о переменных в *makefile*'е как о переменных в традиционных языках программирования, есть существенное отличие между макропеременными и «традиционными» переменными. Значения макропеременных подставляются сразу при встрече их имени в тексте программы, порождая строку, которая затем также сканируется на наличие макросов. Это отличие станет более ясным, когда мы рассмотрим переменные *make* подробнее.

Имена переменных могут содержать почти любые символы, включая многие знаки пунктуации. Разрешаются даже пробелы, но, если вы считаете себя здравомыслящим человеком, избегайте их. Не разрешается использовать в составе имени переменных следующие символы: `:`, `#` и `=`.

Регистр букв в имени переменных имеет значение, то есть переменные `cc` и `CC` являются различными переменными. Для получения значения переменной нужно заключить её имя внутрь круглых скобок, предваряемых символом доллара (`$( )`). Из этого правила есть одно исключение: если имя переменной состоит из од-

ного символа, то круглые скобки можно опустить и писать просто `$символ`. Вот почему автоматически переменные могут использоваться без круглых скобок. Как правило вам стоит предпочитать форму со скобками и избегать переменных, имя которых состоит из одного символа.

Значение переменной также может быть получено с использованием фигурных скобок, например, `${CC}`. Эта форма встречается довольно часто, в частности, в старых *makefile*'ах. Трудно найти причину, по которой использование одной из этих форм было бы предпочтительным. Выберите для себя какую-то одну и придерживайтесь её. Некоторые люди используют фигурные скобки для ссылки на переменную, а круглые — для вызовов функций, подражая синтаксису командного интерпретатора Bourne shell. В современных *makefile*'ах используются круглые скобки, вот почему мы будем придерживаться именно этого стиля в этой книге.

Существуют определённые соглашения относительно имён переменных. Все буквы имени переменных, представляющих значения, не изменяемые в ходе работы *make* (констант), и которые могут быть указаны пользователем через интерфейсы командной строки или переменных окружения, должны быть заглавными. Слова внутри имён таких переменных обычно разделяются подчёркиваниями. Имена переменных, которые встречаются только внутри *makefile*'а, содержат только прописные буквы, слова в них также разделяются подчёркиваниями. Наконец, в этой книге имена всех функций, определяемых пользователем с помощью переменных или макросов, состоят из прописных букв, слова внутри имён разделяются знаками тире. Другие соглашения относительно имён будут оглашаться тогда, когда в этом будет необходимость. Следующие примеры используют функциональность, которую мы ещё не обсуждали. Поскольку они иллюстрируют применение соглашений именования, не старайтесь вникать в детали:

```
# Обычные константы.
CC    := gcc
MKDIR := mkdir -p

# Внутренние переменные.
sources = *.c
objects = $(subst .c,.o,$(sources))

# Пара функций.
maybe-make-dir  = $(if $(wildcard $1),,$(MKDIR) $1)
assert-not-null = $(if $1,,$(error Illegal null value.))
```

Значение переменной состоит из всех слов, находящихся справа от знака присваивания без учёта начального пробела. Пробелы в конце строки также входят в состав значения. Иногда это может вызывать проблемы, например, при использовании переменных, чьи значения оканчиваются пробелами, в сценариях командного интерпретатора:

```
LIBRARY = libio.a # LIBRARY заканчивается пробелом

missing_file:
    touch $(LIBRARY)
    ls -l | grep '$(LIBRARY)'
```

Присваивание переменной содержит пробел, который становится более заметным за счёт комментария (однако на самом деле комментария может и не быть). После запуска *make* мы увидим следующий вывод:

```
$ make

touch libio.a
ls -l | grep 'libio.a '
make: *** [missing_file] Error 1
```

Поскольку шаблон поиска, переданный программе *grep*, также содержит пробел, поиск его вхождения в выводе команды *ls* закончился неудачей. Позднее мы обсудим проблемы, связанные с пробелами, более детально. А пока давайте рассмотрим поближе переменные *make*.

3.1 Для чего используются переменные

В целом использование переменных для представления внешних программ является хорошей идеей, поскольку позволяет пользователю легко адаптировать *makefile* под своё окружение. Например, в системе часто бывает несколько версий *awk*: *awk*, *nawk* и *gawk*. Создавая переменную *AWK* для хранения имени программы *awk*, вы облегчаете жизнь пользователям вашего *makefile*'а. К тому же, если безопасность критична для вашей среды, доступ к внешним программам с указанием абсолютного пути к их исполняемым файлам является хорошей практикой решения проблем с переменной `PATH` пользователя. Абсолютные пути также уменьшают вероятность запуска троянского коня под видом системной утилиты. И, конечно, абсолютные пути делают ваши *makefile*'ы менее переносимым. Здесь вы должны сделать выбор, основываясь на ваших собственных требованиях.

Хотя наше первое использование переменных заключалось в хранении простых констант, переменные также могут быть использованы для хранения последовательностей команд, например, следующий пример определяет функцию для вывода отчёта о количестве свободных блоков в файловой системе[1]:

[1]Команда *df* возвращает список всех смонтированных файловых систем, их ёмкость и объём занятого пространства. Если задан аргумент, то выводится статистика для указанной файловой системы. Первая строка вывода содержит заголовки столбцов. Вывод этой команды подаётся на вход сценария *awk*, извлекающего вторую строку и игнорирующего остальные. Четвёртый столбец в выводе *df* — число свободных блоков.

```
DF  = df
AWK = awk
free-space := $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
```

Переменные могут использоваться для обоих указанных целей. Как мы увидим позднее, это не единственные их применения.

3.2 Типы переменных

В *make* существует два типа переменных: упрощённо вычисляемые (simple expanded variables) и рекурсивно вычисляемые (recursively expanded variables). *Упрощённо вычисляемые* переменные (или *простые переменные*) определяются при помощи оператора присваивания «`:=`»:

```
MAKE_DEPEND := $(CC) -M
```

Такие переменные называются «упрощённо вычисляемыми» потому, что правая часть присваивания вычисляется непосредственно при чтении *makefile*'а. При этом подставляется значение всех переменных *make*, входящих в правую часть, и результирующий текст сохраняется в качестве значения переменной. Это поведение идентично поведению большинства языков программирования и командных сценариев. Например, вычисление предыдущей переменной, скорее всего, породит текст `gcc -M`. Однако если переменная `CC` не определена, то переменная `MAKE_DEPEND` примет значение `<пробел>-M`. В этом случае выражение `$(CC)` вычисляется как пустая строка, поскольку переменная `CC` ещё не определена. Отсутствие определения переменной не является ошибкой. На самом деле это очень удобно. Большинство неявных правил содержат неопределённые переменные, необходимые для настройки поведения правил пользователями. Если пользователь не определяют никаких настроек, переменные просто содержат пустые строки. Теперь рассмотрим пробел в начале полученного значения. Правая часть присваивания после отбрасывания начального пробела выглядит следующим образом: `$CC -M`. После того, как ссылка на переменную вычисляется как пустая строка, *make* не производит повторного сканирования и не удаляет начальные пробелы.

Рекурсивно вычисляемые переменные (или просто рекурсивные переменные) определяются при помощи оператора присваивания «`=`»:

```
MAKE_DEPEND = $(CC) -M
```

Второй тип переменных называется «рекурсивно вычисляемые переменные» потому, что правая часть присваивания просто копируется *make* и сохраняется как значение переменной без вычисления. Вместо этого вычисление происходит каждый раз, когда переменная *используется*. Быть может, более подходящим названием для таких переменных — *лениво вычисляемые* переменные, поскольку

вычисления откладываются до тех пор, пока не потребуется их результат. Одним из удивительных следствий такого рода вычислений заключается в том, что присваивания могут осуществляться «в неправильном порядке»:

```
MAKE_DEPEND = $(CC) -M
...
# Чуть позже
CC = gcc
```

Теперь значение `MAKE_DEPEND` будет равно `gcc -M`, несмотря на то что значение `CC` не определено в момент присваивания значения переменной `MAKE_DEPEND`.

На самом деле рекурсивные переменные не являются просто ленивыми присваиваниями (по крайней мере, обычными ленивыми присваиваниями). Каждый раз при обращении к рекурсивной переменной её значение вычисляется заново. Для переменных, определённых в терминах простых констант, таких как `MAKE_DEPEND`, эта разница бессмысленна, поскольку все переменные справа от оператора присваивания также являются простыми константами. Однако представим, что переменная в присваиваемом выражении представляет собой результат выполнения некоторой программы, например, *date*. Каждый раз, когда подобная рекурсивная переменная будет вычисляться, будет происходить запуск программы *date* и переменная будет получать новое значение (в предположении, что повторное вычисление происходит по крайней мере через секунду). Иногда это чрезвычайно полезно. А иногда весьма раздражает!

Другие виды присваивания

В предыдущем примере мы видели два типа присваивания: «`=`» для определения рекурсивных переменных и «`:=`» для определения простых переменных. *make* имеет ещё два оператора присваивания.

Оператор «`?=`» называется *оператором условного присваивания переменной*. Для краткости мы будем называть его просто условным присваиванием. Этот оператор осуществляет присваивание переменной только в том случае, если её значение ещё не определено.

```
# Положить полученные файлы в каталог $(PROJECT_DIR)/out.
OUTPUT_DIR ?= \(PROJECT_DIR)/out
```

В этом примере мы присвоим значение переменной `OUTPUT_DIR` только в том случае, если оно ещё не было определено. Такое поведение очень удобно для работы с переменными окружения. Мы обсудим этот вопрос более подробно в разделе «Где определяются переменные».

Другой оператор присваивания, +=, обычно называют *добавлением*. Как можно предположить из названия, этот оператор добавляет текст к значению переменной. Может быть не очевидно, что это довольно важная функциональность, необходимая при использовании рекурсивных переменных. Значение справа от этого оператора присваивания добавляется к значению переменной, *не изменяя в переменной первоначального значения*. «Подумаешь,» — скажете вы, — «Разве не так обычно работает добавление?». Да, но здесь есть одна маленькая хитрость.

Добавление текста к простой переменной реализуется очевидным образом. Оператор += может быть реализован так:

```
простая_переменная := $(простая_переменная) что-то ещё
```

Поскольку значение простой переменной уже было вычислено, *make* может просто вычислить выражение `$(простая_переменная)`, добавить требуемый текст и закончить присваивание. Но рекурсивные переменные порождают проблему. Следующая реализация недопустима:

```
рекурсивная_переменная = $(рекурсивная_переменная) что-то ещё
```

Это выражение является ошибкой, потому что для *make* не существует корректного способа его обработки. Если *make* сохранит текущее значение рекурсивной переменной плюс текст «что-то ещё», то не сможет вычислить правильное значение позднее. Более того, попытка вычислить рекурсивную переменную, содержащую ссылку на себя, приводит к бесконечному циклу:

```
$ make

makefile:2: *** Recursive variable ‘recursive’ references
itself (eventually).  Stop.
```

Таким образом, оператор «+=» был создан специально для возможности добавления текста к рекурсивным переменным. В частности, этот оператор полезен при инкрементом определении значения переменной.

3.3 Макросы

Переменные хороши для хранения одиночных строк текста, но что если мы хотим сохранить многострочное значение, такое, как командный сценарий, который мы хотим выполнять в нескольких правилах? Например, следующая последовательность команд может быть использована для создания Java архива (*jar*) из *.class* файлов:

```
echo Creating $@...
$(RM) $(TMP_JAR_DIR)
$(MKDIR) $(TMP_JAR_DIR)
$(CP) -r $^ $(TMP_JAR_DIR)
cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
$(JAR) -ufm $@ $(MANIFEST)
$(RM) $(TMP_JAR_DIR)
```

В начале длинной последовательности (наподобие предыдущей) я предпочитаю печатать диагностическое сообщение. Это значительно упрощает чтение вывода *make*. После вывода сообщения мы помещаем наши *.class* файлы в новый временный каталог, удалив сначала этот каталог, если он существует[2], затем создав новый. После этого мы копируем в этот каталог каталоги-реквизиты (и все их подкаталоги). Затем мы переходим в него и создаём jar-архив с именем цели. Наконец, мы добавляем к архиву файл манифеста и осуществляем удаление временного каталога. Естественно, мы не хотим делать копий этой последовательности команд, поскольку в будущем это может усложнить поддержку. Мы можем рассмотреть вариант упаковки всех этих команд в рекурсивную переменную, но такой подход неудобен при поддержке и труден для чтения при выводе *make* (вся последовательность команд будет выведена на экран как одна большая строка текста).

Вместо этого мы можем использовать «упакованную последовательность команд» GNU *make*, созданную при помощи директивы `define`. Термин «упакованная последовательность» немного неуклюж, поэтому мы будем называть это *макросом*. На самом деле макросы — это просто ещё один метод определения переменных в *make*, позволяющий помещать символы окончания строки внутрь значения переменной. Руководство пользователя GNU *make* использует слова *переменная* и *макрос* как синонимы. В этой книге мы будем использовать термин *макрос* исключительно для обозначения переменных, определённых с помощью директивы `define`, если же определение происходит при помощи оператора присваивания, будет применяться термин *переменная*.

```
define create-jar
  @echo Creating $@...
  $(RM) $(TMP_JAR_DIR)
  $(MKDIR) $(TMP_JAR_DIR)
  $(CP) -r $^ $(TMP_JAR_DIR)
  cd $(TMP_JAR_DIR) && $(JAR) $(JARFLAGS) $@ .
  $(JAR) -ufm $@ $(MANIFEST)
  $(RM) $(TMP_JAR_DIR)
endef
```

[2]Для достижения наилучшего эффекта переменная `RM` должна иметь значение `rm -rf`. Как правило, по умолчанию она определена как `rm -f`, что безопаснее, но не так эффективно. Переменная `MKDIR` должна иметь значение `mkdir -p`, и так далее (прим. автора).

За директивой `define` следуют имя макроса и новая строка. Тело макроса содержит весь текст вплоть до слова `endef`, которое должно находиться в отдельной строке. Макросы вычисляются практически также, как и другие переменные, за тем исключением, что в контексте командного сценария в начало каждой строки добавляется символ табуляции. Вот пример, использующий эту особенность:

```
$(UI_JAR): $(UI_CLASSES)
    $(create-jar)
```

Обратите внимание на символ `@`, добавленный перед командой `echo`. *make* не выводит команды с таким префиксом перед их выполнением. Когда мы запустим *make*, мы не увидим в выводе команды `echo`, только результат её выполнения. Если префикс `@` применяется внутри макроса, то его действие распространяется только на ту строку, которую он предваряет. Однако, если применить его при вызове макроса, его действие будет распространятся на всё тело макроса:

```
$(UI_JAR): $(UI_CLASSES)
    @$(create-jar)
```

Одним из результатов выполнения предыдущего примера будет следующий вывод:

```
$ make
Creating ui.jar...
```

Использование префикса `@` рассматривается более детально в разделе «Модификаторы команд» главы 5.

3.4 Когда переменные получают свои значения

В предыдущем разделе мы начали рассматривать «закулисье» вычисления значения переменных. Результат во многом зависит от того, что уже было определено, и где это было определено. Вы можете получить неожиданный результат, даже если *make* не может найти ошибки в вашей спецификации. Так каковы же правила вычисления переменных? Как на самом деле всё это работает?

После запуска *make* выполняет свою работу в две фазы. Во время первой фазы *make* читает *makefile* и все включаемые им файлы. На этом этапе переменные и правила загружаются во внутреннюю базу данных *make*, после чего создаётся граф зависимостей. Во время второй фазы *make* анализирует граф зависимостей и определяет цели, которые нуждаются в сборке, затем выполняет командные сценарии для сборки реквизитов.

Когда *make* встречает директиву `define` или определение рекурсивной переменной, строки значения переменной или тела макроса сохраняются вместе с символами новой строки без каких-либо вычислений. Самый последний символ новой

Определение	***A* вычисляется**	***B* вычисляется**
$A = B$	Сразу	При использовании
$A? = B$	Сразу	При использовании
$A := B$	Сразу	Сразу
$A+ = B$	Сразу	Сразу или при использовании

`define A`
`B ...`
`B ...`
`endef`
	Сразу	При использовании

Таблица 3.1: Правила для незамедлительного и отложенного вычислений

строки в теле макроса не сохраняется в тексте определения. Иначе при вычислении макроса читался бы один лишний символ новой строки.

При вычислении макроса полученный текст сразу же сканируется на содержание других макросов или переменных, подлежащих вычислению, этот процесс продолжается рекурсивно. Если макрос вычисляется в контексте командного сценария, в начало каждой строки добавляется символ табуляции.

Итак, вычисление переменных и макросов *make* подчиняется следующим правилам:

- В случае присваивания переменной значения левая часть присваивания всегда вычисляется сразу на первой фазе работы *make*.
- Вычисление правой части операторов `=` и `?=` откладывается до тех пор, пока не потребуется значение соответствующей переменной.
- Правая часть оператора `:=` вычисляется сразу.
- Правая часть оператора `+=` вычисляется сразу, если переменная в левой части изначально была определена как простая, иначе вычисление откладывается.
- В определении макроса (использующего директиву `define`) имя определяемого макроса вычисляется сразу, вычисление тела макроса откладывается.
- Имена целей и реквизитов всегда вычисляются сразу, вычисление команд всегда откладывается.

Таблица 3.1 содержит правила порядка вычисления выражений при определении переменных.

Примите за правило определять переменные и макросы перед их использованием. В частности, требуется, чтобы переменная, используемая в описании цели или реквизита была определена.

Думаю, пример многое прояснит. Предположим, мы решили переделать наш макрос `free-space`. Сначала рассмотрим отдельные части примера, затем соберём всё вместе.

```
BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

Мы определяем три переменных, содержащие имена программ, которые будут использоваться в нашем макросе. Поскольку все переменные являются простыми, их значения должны быть вычислены во время чтения *makefile*'а. Так как переменная `BIN` определена раньше остальных, её значение может быть использовано в определениях других переменных.

Далее определим макрос `free-space`.

```
define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef
```

За директивой `define` следует имя переменной, которое сразу вычисляется. В нашем случае вычисления не требуется. Тело макроса считывается и сохраняется не вычисленным.

Наконец, используем наш макрос внутри правила.

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
    $(free-space)
```

Когда считывается цель `$(OUTPUT_DIR)/very_big_file`, происходит подстановка значений всех переменных. Значение выражения `$(OUTPUT_DIR)` вычисляется как `/tmp`, формируя цель `/tmp/very_big_file`. Затем считывается командный сценарий, ассоциированный с этой целью. Строки с командами распознаются благодаря наличию символа табуляции, считываются и сохраняются, но подстановка значений переменных и макросов не происходит.

Теперь соберём отрывки воедино. Изменим порядок их следования для иллюстрации алгоритма вычисления *make*:

```
OUTPUT_DIR := /tmp

$(OUTPUT_DIR)/very_big_file:
    $(free-space)
```

```
define free-space
  $(PRINTF) "Free disk space "
  $(DF) . | $(AWK) 'NR == 2 { print $$4 }'
endef

BIN    := /usr/bin
PRINTF := $(BIN)/printf
DF     := $(BIN)/df
AWK    := $(BIN)/awk
```

Заметим, что несмотря на то, что порядок строк кажется обратным, выполнение происходит успешно. В этом заключается один из замечательных эффектов рекурсивных переменных. Они могут быть невероятно полезны и совершенно непонятны одновременно. Наш *makefile* работает как нужно благодаря тому, что вычисление командных сценариев и тел макросов откладываются до тех пор, пока результаты этих вычислений не потребуются. Таким образом, порядок, в котором появляются определения, не влияет на выполнение.

На второй фазе выполнения, когда *makefile* уже прочитан, *make* определяет цели, анализирует граф зависимостей и выполняет действия, ассоциированные с каждым правилом. Поскольку мы специфицировали только одну цель, не имеющую реквизитов (`$(OUTPUT_DIR)/very_big_file`), *make* просто выполнит действия, с ассоциированные с ней (предположим, такой файл не существует) — команду `$(free-space)`. После вычислений *make* получит следующее:

```
/tmp/very_big_file:
    /usr/bin/printf "Free disk space "
    /usr/bin/df . | /usr/bin/awk 'NR == 2 { print $$4 }'
```

Как только значения всех переменных вычислены, *make* выполняет команды одну за другой. Давайте рассмотрим две части *makefile*'а, в которых порядок имеет значение. Как упоминалось ранее, имя цели `$(OUTPUT_DIR)/very_big_file` вычисляется сразу. Если бы определение переменной `OUTPUT_DIR` находилось после спецификации правила, результатом вычисления имени стала бы строка `/very_big_file`. Скорее всего, это не то, чего хотел пользователь. Если бы определение `BIN` было помещено после определения `AWK`, наши переменные получили бы значения `/printf`, `/df` и `/awk`, так как оператор `:=` вызывает немедленное вычисление правой части присваивания. Однако в этом случае мы можем избежать проблемы, использовав для определения переменных `PRINTF`, `DF` и `AWK` оператор `=` вместо оператора `:=` и сделав тем самым эти переменные рекурсивными.

Наконец, обратите внимание на одну деталь. Объявление переменных `OUTPUT_DIR` и `BIN` как рекурсивных не решило бы рассмотренных проблем порядка спецификаций. Важно здесь то, что в момент вычисления значения переменной `$(OUTPUT_DIR)/very_big_file` и правых частей определений `PRINTF`, `DF` и `AWK` значения переменных, на которые ссылаются эти выражения, должны быть уже определены.

3.5 Переменные, зависящие от цели или шаблона

Обычно переменные принимают только одно значение во время выполнения *make*. Это гарантируется двухэтапной обработкой *makefile*'а. На первом этапе *make* читает *makefile*, производит присваивание и вычисление переменных и строит граф зависимостей. На втором этапе производится анализ графа зависимостей. Таким образом, когда происходит выполнение команд, обработка переменных уже закончена. Предположим, однако, что нам нужно переопределить значение переменной только для какой-то цели или шаблона.

В приведённом ниже примере файл, который мы компилируем, требует использования дополнительной опции `-DUSE_NEW_MALLOC=1`, которая не должна быть использована при компиляции других файлов:

```
gui.o: gui.h
    $(COMPILE.c) -DUSE_NEW_MALLOC=1 $(OUTPUT_OPTION) $<
```

Проблема решена добавлением дубликата правила компиляции, включающего необходимую опцию. Такой подход неудовлетворителен по нескольким причинам. Во-первых, мы повторяем код. Если правило когда-нибудь изменится, или если мы решим заменить встроенное правило собственным, этот код потребует изменения, о котором легко можно забыть. Во-вторых, если множество файлов требует специализированных опций, копирование и вставка кусков кода быстро становится утомительным и рискованным занятием (представьте, что у вас сотни таких файлов).

Для решения таких проблем *make* предоставляет функциональность *переменных, зависящих от цели*. Определения этих переменных привязаны к цели и действительны только во время обработки цели или её реквизитов. Используя эту функциональность, мы можем переписать наш пример следующим образом:

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
$(COMPILE.c) $(OUTPUT_OPTION) $<
```

Переменная `CPPFLAGS` встроена в стандартное правило компиляции исходных файлов C и предназначена для опций препроцессора C. При помощи оператора `+=` мы добавляем новую опцию к уже существующим. Теперь сценарий компиляции может быть удалён полностью:

```
gui.o: CPPFLAGS += -DUSE_NEW_MALLOC=1
gui.o: gui.h
```

Пока происходит сборка цели `gui.o`, значение переменной `CPPFLAGS` вдобавок к оригинальному значению будет содержать строку `-DUSE_NEW_MALLOC=1`. Когда цель *gui.o* будет собрана, значение `CPPFLAGS` будет восстановлено.

Общий синтаксис определения переменных, зависящих от цели, таков:

```
цель ...: переменная  = значение
цель ...: переменная := значение
цель ...: переменная += значение
цель ...: переменная ?= значение
```

Как вы могли заметить, для определения таких переменных применимы все формы оператора присваивания. Переменная не обязательно должна существовать до присваивания.

Более того, присваивание переменным значения не осуществляется до начала сборки цели. Таким образом, правая часть присваивания может содержать ссылку на значение другой переменной, зависящей от этой цели. Переменная также действительна во время сборки всех реквизитов.

3.6 Где определяются переменные

Пока все переменные в наших *makefile*'ах определялись явно. На самом деле они могут иметь и более сложное происхождение. Например, мы уже видели, что переменные могут определяться через интерфейс командной строки *make*. На самом деле переменные могут определяться в следующих источниках:

Файл

Естественно, переменные могут быть определены в *makefile*'е или в файле, им подключенным.

Командная строка

Переменные могут быть определены или переопределены прямо из командной строки *make*:

```
$ make CFLAGS=-g CPPFLAGS='-DBSD -DDEBUG'
```

Аргументы командной строки, содержащие символ `=`, являются определениями переменных. Каждое такое определение должно быть отдельным аргументом. Если значение переменной (или, упаси вас Боже, её имя) содержит пробелы, нужно либо экранировать пробел, либо заключить этот аргумент в кавычки.

Значение, полученное переменной через интерфейс командной строки, переопределяет любое другое определение, сделанное в *makefile*'е или содержащееся в окружении. Присваивания в командной строке могут определять как простые, так и рекурсивные переменные с помощью операторов `:=` и `=` соответственно. Однако всё же существует возможность повысить приоритет определения переменной в *makefile*'е, для этого нужно использовать директиву `override`.

```
# Для успешной компоновки необходим порядок байтов от старшего к
# младшим.
override LDFLAGS = -EB
```

Разумеется, игнорировать явные определения пользователя стоит только в чрезвычайных обстоятельствах.

Окружение

После старта *make* все переменные окружения автоматически становятся переменными *make*. Эти переменные имеют очень низкий приоритет, поэтому присваивание значения этим переменным, сделанное в *makefile*'е или через командную строку, переопределит переменную окружения. Вы можете форсировать переопределение переменных в *makefile*'е переменными окружения при помощи опции `--environment-overrides` (или просто `-e`).

Когда *make* вызывается рекурсивно, некоторые переменные из родительского процесса *make* передаются через окружение дочернему процессу. По умолчанию только переменные, изначально определённые в окружении, попадают в окружение дочернего процесса. Однако любая переменная может быть помещена в окружение с помощью директивы `export`:

```
export CLASSPATH := \$(HOME)/classes:\$(PROJECT)/classes
SHELLOPTS = -x
export SHELLOPTS
```

Вы также можете экспортировать все переменные:

```
export
```

Обратите внимание на то, что *make* может экспортировать даже те переменные, имена которых содержат недопустимые с точки зрения командного интерпретатора символы. Например, результатом запуска следующего *makefile*'а:

```
export valid-variable-in-make = Neat!
show-vars:
    env | grep '^valid-'
    valid_variable_in_shell=Great
    invalid-variable-in-shell=Sorry
```

будет вывод:

```
$ make
env | grep '^valid-'
valid-variable-in-make=Neat!
valid_variable_in_shell=Great
invalid-variable-in-shell=Sorry
/bin/sh: line 1: invalid-variable-in-shell=Sorry: command not found
make: *** [show-vars] Error 127
```

«Недопустимая» переменная командного интерпретатора была создана при помощи экспорта из *make* переменной `valid-variable-in-make`. Эта переменная не будет доступна стандартными средствами интерпретатора, только при помощи трюков наподобие применения программы *grep* ко всему окружению. Тем не менее, эта переменная будет доступна в дочернем процессе *make*. Мы рассмотрим применение рекурсивных вызовов *make* во второй части книги.

Вы также можете запретить экспорт переменной в окружение дочернего процесса:

```
unexport DISPLAY
```

Директивы `export` и `unexport` работают так же, как и их аналоги из интерпретатора *sh*.

Оператор условного присваивания очень удобен при работе с переменными окружения. Допустим, вы определили в своём *makefile*'е стандартный каталог для хранения создаваемых файлов и хотите предоставить пользователю возможность легко его изменить. Условное присваивание идеально подходит для таких ситуаций:

```
# Пусть каталогом по умолчанию будет $(PROJECT_DIR)/out.
OUTPUT_DIR ?= $(PROJECT_DIR)/out
```

В этом случае присваивание произойдёт только в том случае, если значение переменной `OUTPUT_DIR` ещё не определено. Мы можем добиться того же эффекта более «многословным» способом:

```
ifndef OUTPUT_DIR
  # Пусть каталогом по умолчанию будет $(PROJECT_DIR)/out.
  OUTPUT_DIR = $(PROJECT_DIR)/out
endif
```

Разница между этими двумя определениями заключается в том, что оператор условного присваивания не будет выполнен, если переменная `OUTPUT_DIR`

определена, пусть даже имеет пустое значение, тогда как операторы `ifdef` и `ifndef` проверяют свой аргумент на пустоту. Таким образом, выражение `OUTPUT_DIR=` с точки зрения оператора условного присваивания будет считаться определением, а с точки зрения оператора `ifdef` — нет.

Следует отметить, что чрезмерное использование переменных окружения делает ваши *makefile*'ы менее переносимыми, поскольку разные пользователи, как правило, имеют разное окружение. На самом деле, я редко использую переменные окружения именно по этой причине.

Автоматические переменные

Наконец, *make* создаёт автоматические переменные непосредственно перед выполнением сценария, ассоциированного с правилом.

Традиционно переменные окружения используются для упрощения управления разницей в настройке машин разработчиков. Например, общей практикой является создание среды разработки (инструментов разработки, исходного кода и дерева полученных компиляцией файлов) на основании переменных окружения, используемых в *makefile*'е. При таком подходе *makefile* определяет одну переменную для корня каждого дерева каталогов. Если корень дерева исходных файлов содержится в переменной `PROJECT_SRC`, корень дерева объектных файлов — в переменной `PROJECT_BIN`, а каталог библиотек — в переменной `PROJECT_LIB`, то разработчик вправе определить эти каталоги так, как он считает нужным.

Потенциальной проблемой такого подхода (и вообще использования переменных окружения) является ситуация, когда упомянутые выше переменные не определены. Одним из решений является определение значений по умолчанию в *makefile*'е с помощью оператора условного присваивания:

```
PROJECT_SRC ?= /dev/$(USER)/src
PROJECT_BIN ?= $(patsubst %/src,%/bin,$(PROJECT_SRC))
PROJECT_LIB ?= /net/server/project/lib
```

Используя эти переменные для доступа к компонентам проекта, вы можете создать среду разработки, адаптируемую под различные настройки машин разработчиков (мы увидим более развёрнутые примеры во второй части книги). Однако опасайтесь чрезмерного использования переменных окружения. Как правило, *makefile* следует составлять так, чтобы как можно меньше использовать окружение разработчика, предоставлять разумные настройки по умолчанию и проверять наличие критически важных компонентов.

3.7 Условная обработка и включения

Части *makefile*'а могут быть опущены или выбраны для обработки во время его чтения с помощью директив *условной обработки*. Условия, контролирующие

обработку, могут принимать несколько форм, таких как «A определено» или «A равно B». Например:

```
# переменная COMSPEC определена только в Windows.
ifdef COMSPEC
  PATH_SEP := ;
  EXE_EXT  := .exe
else
  PATH_SEP := :
  EXE_EXT  :=
endif
```

В предыдущем примере обработается первая ветвь условия только в том случае, если переменная `COMSPEC` определена. Синтаксис директив условной обработки имеет две формы:

```
if-условие
  текст для обработки если условие выполнено
endif
```

и:

```
if-условие
  текст для обработки если условие выполнено
else
  текст для обработки если условие не выполнено
endif
```

Значения шаблона *if-условие* могут быть следующими:

```
ifdef  имя-переменной
ifndef имя-переменной
ifeq  тест
ifneq тест
```

При использовании директив `ifdef\ifndef` *имя-переменной* не нужно заключать в скобки (`$( )`). Наконец, значением шаблона *тест* может быть одно из следующих выражений:

```
"a" "b"
(a,b)
```

В выражениях могут использоваться двойные или одинарные кавычки по желанию (однако тип открывающейся и закрывающейся кавычки должен совпадать).

Директивы условной обработки могут быть использованы внутри тела макросов или в командных сценариях:

```
libGui.a: $(gui_objects)
    $(AR) $(ARFLAGS) $@ $<
  ifdef RANLIB
    $(RANLIB) $@
  endif
```

Я предпочитаю делать отступ перед директивами условной обработки, однако неосторожное выравнивание может привести к ошибкам. В предыдущем примере перед директивами сделан отступ в два пробела, а заключённые в них команды предваряются символом табуляции. *make* не может распознать команды, не начинающиеся с символа табуляции. Если директива условной обработки предваряется символом табуляции, она рассматривается как команда и передаётся в командный интерпретатор.

Директивы `ifeq` и `ifneq` проверяют свои аргументы на на равенство и неравенство соответственно. Пробелы в условиях директив могут быть причиной трудноуловимых ошибок. Например, когда используется форма теста с круглыми скобками, пробел после запятой не учитывается, тогда как все остальные пробелы имеют значение:

```
ifeq (a, a)
  # Равенство
endif

ifeq ( b, b )
  # Неравенство - ‘ b’ != ‘b ’
endif
```

Лично я предпочитаю форму с кавычками:

```
ifeq "a" "a"
  # Равенство
endif

ifeq ’b’ ’b’
  # Тоже равенство
endif
```

Однако иногда случается так, что значение переменной содержит пробел в начале или в конце. Это может быть источником ошибки, поскольку сравнение учитывает все символы. Для создания более надёжных *makefile*’ов используйте функцию *strip*:

```
ifeq "$(strip $(OPTIONS))" "-d"
  COMPILATION_FLAGS += -DDEBUG
endif
```

3.7.1 Директива `include`

Мы уже встречали директиву `include` в разделе «Автоматическое определение зависимостей» главы 2. Теперь давайте рассмотрим её более детально.

Любой *makefile* может включать другие файлы. Наиболее частое использование этой возможности — помещение общих определений *make* в заголовочный файл и включение автоматически составленных файлов зависимостей. Директива `include` используется следующим образом:

```
include definitions.mk
```

Параметры директивы могут содержать произвольное число файлов, шаблоны командного интерпретатора и переменные *make*.

3.7.2 Директива `include` в контексте зависимостей

Когда *make* встречает директиву `include`, он производит раскрытие шаблонов и подстановку значений переменных, а затем пытается прочитать подключенные файлы. Если файл существует, выполнение продолжается. Если же указанный файл не существует, *make* выводит предупреждение и продолжает читать остаток *makefile*'а. Когда весь *makefile* прочитан, *make* просматривает базу данных в поисках правила сборки подключаемых файлов. Если соответствие найдено, *make* следует найденному правилу сборки цели. Если хотя бы один из включаемых файлов был собран, *make* очищает свою базу данных и производит повторное чтение всего *makefile*'а. Если после завершения всего процесса чтения, сборки и повторного чтения какая-то из директив `include` завершается неудачей ввиду отсутствующих файлов, *make* завершает своё выполнение с ненулевым кодом возврата.

Мы можем увидеть этот процесс в действии при помощи следующего примера, состоящего из двух файлов. Мы используем встроенную функцию *warning*, для вывода сообщений из *make* (эта и многие другие функции рассмотрены в главе 4). Вот *makefile*:

```
# Простой makefile, включающий файл.
include foo.mk
$(warning Finished include)

foo.mk: bar.mk
    m4 --define=FILENAME=$@ bar.mk > $@
```

Ниже представлен *bar.mk*, подключаемый в *makefile*:

```
# bar.mk - выдать сообщение о чтении файла.
$(warning Reading FILENAME)
```

После запуска *make* мы увидим следующий вывод:

```
$ make
Makefile:2: foo.mk: No such file or directory
Makefile:3: Finished include
m4 --define=FILENAME=foo.mk bar.mk > foo.mk
foo.mk:2: Reading foo.mk
Makefile:3: Finished include
make: 'foo.mk' is up to date.
```

Первая строка отражает тот факт, что *make* не смог найти включаемый файл, однако, вторая строка показывает, что чтение и выполнение *makefile*'а продолжилось. По завершении чтения *make* обнаружил правило для создания подключаемого файла, *foo.mk*, и выполнил соответствующий сценарий. Затем *make* начал весь процесс заново, в этот раз не встретив никаких трудностей с чтением включаемого файла.

Теперь самое время заметить, что *make* интерпретирует *makefile* как потенциальную цель. После того, как *makefile* прочитан, *make* ищет правило для сборки текущего *makefile*'а. Если такое правило находится, *make* выполняет соответствующий правилу сценарий и проверяет, изменился ли текущий *makefile*. Если *makefile* изменился, *make* производит очистку своего состояния и считывает *makefile* заново, повторяя анализ. Ниже приведён простой пример бесконечного цикла, основанный на описанном поведении:

```
.PHONY: dummy

makefile: dummy
    touch $@
```

Когда *make* выполняет *makefile*, он видит, что файл нуждается в сборке (поскольку цель `dummy` является абстрактной) и выполняет команду *touch*, которая изменяет время последней модификации *makefile*'а. Затем *make* читает файл и обнаруживает, что он требует обновления.... В общем, вы поняли.

Где *make* ищет включаемые файлы? Если аргумент директивы `include` является абсолютным путём, то *make* открывает файл по указанному пути. Если указан относительный путь, *make* ищет файл относительно текущего рабочего каталога. Если файл не найден, то осуществляется поиск в каталогах, указанных при помощи опции `--include-dir` (или просто `-I`). Если файл не найден и там, производится поиск в стандартных каталогах, указанных при компиляции *make*: */usr/local/include*, */usr/gnu/include*, */usr/include*. Пути могут отличаться, поскольку они зависят от опций компиляции *make*.

Если *make* не может найти файл и не может собрать его с помощью правила, происходит выход с ненулевым кодом возврата. Если вы хотите, чтобы *make* игнорировал включение несуществующих файлов, добавьте знак дефиса перед директивой `include`:

```
-include i-may-not-exist.mk
```

Для обратной совместимости с другими версиями *make* слово `sinclude` является синонимом `-include`.

3.8 Стандартные переменные *make*

Вдобавок к автоматическим переменным, *make* содержит переменные, предназначенные для получения текущего состояния *make* и настройки встроенных правил:

`MAKE_VERSION`

Значением этой переменной является номер текущей версии GNU *make*. Во время написания этой книги этим значением было `3.80`, а CVS хранилище содержало версию `3.81rc1`.

Предыдущая версия *make*, `3.79.1`, не поддерживающая функций *eval* и *value*, имеет довольно широкое распространение. Так что когда пишу свои *makefile*'ы, требующие подобной функциональности, я использую эту переменную для проверки версии *make*. Мы увидим примеры такого использования в разделе «Функции управления выполнением» главы 4.

`CURDIR`

Эта переменная содержит текущий рабочий каталог (current working directory, cwd) исполняемого процесса *make*. Это тот же самый каталог, в котором была выполнена команда *make* (и тот же самый каталог, что хранится в переменной командного интерпретатора `PWD`), если только не использовалась опция `--directory` (`-C`). Опция `--directory` сообщает *make*, что нужно изменить текущий каталог перед тем, как начать искать *makefile*. Полная форма опции выглядит следующим образом: `--directory=`*`имя-каталога`* или `-C` *`имя-каталога`*. Если использовалась опция `--directory`, то переменная `CURDIR` будет содержать аргумент этой опции.

Обычно я вызываю *make* из редактора *emacs* во время редактирования кода. Например, мой текущий проект написан на Java и использует единственный *makefile* в корневом каталоге проекта (не обязательно в каталоге, содержащем исходный код). В моём случае использование опции `--directory` позволяет мне запускать *make* из любого каталога с исходными файлами и иметь доступ к *makefile*'у. Внутри *makefile*'а все пути являются относительными и вычисляются от каталога, в котором располагается *makefile*. Абсолютные пути используются очень редко, доступ к ним осуществляется при помощи переменной `CURDIR`.

`MAKEFILE_LIST`

Значение этой переменной содержит список всех файлов, прочитанных *make*, включая стандартный *makefile*, все *makefile*'ы, указанные в опциях командной строки, и файлы, подключённые директивой `include`. Перед чтением каждого файла *make* добавляет его имя к значению переменной `MAKEFILE_LIST`. Итак, *makefile* всегда может определить собственное имя, прочитав последнее слово в этом списке.

`MAKECMDGOALS`

Значение `MAKECMDGOALS` содержит список всех целей, указанных в командной строке для текущего процесса *make*. Оно не включает опций и определений переменных. Например:

```
$ make -f- FOO=bar -k goal <<< 'goal:;# $(MAKECMDGOALS)'
# goal
```

Предыдущий пример использует возможность *make* читать *makefile* из стандартного потока ввода, включающуюся опцией `-f-` (или `--file`). Происходит перенаправление стандартного потока ввода на командную строку с помощью синтаксиса командного интерпретатора *bash* `<<<`[3].

Сам *makefile* состоит из спецификации цели по умолчанию `goal`. Командный сценарий указан на той же строке и отделяется от определения цели точкой с запятой. Сценарий состоит из одной строки:

```
# $(MAKECMDGOALS)
```

Переменная `MAKECMDGOALS` обычно используется в том случае, когда цель требует особой обработки. Выразительным примером является цель `clean`. Когда происходит сборка `clean`, *make* не должен осуществлять стандартную проверку изменения подключаемых файлов (обсуждавшуюся в разделе «Условная обработка и включения» главы 3). Для подавления этой проверки можно применить директиву `ifneq` и переменную `MAKECMDGOALS`:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(subst .xml,.d,$(xml_src))
endif
```

[3]Если вы хотите выполнить этот пример в другом интерпретаторе, наберите:

```
$ echo 'goal:;# $(MAKECMDGOALS)' | make -f- FOO=bar -k goal
```

`.VARIABLES`

Значение этой переменной содержит имена всех переменных, определённых . в *makefile*'е на данный момент, исключая переменные, зависящие от цели. Эта переменная доступна только для чтения и любое её переопределение игнорируется.

```
list:
    @echo "$(.VARIABLES)" | tr ' ' '\015' | grep MAKEF

$ make
MAKEFLAGS
MAKEFILE_LIST
MAKEFILES
```

Как вы уже видели, переменные также используются для настройки встроенных неявных правил *make*. Правила для компиляции и компоновки исходных файлов C/C++ демонстрируют типичную форму, принимаемую переменными для произвольных языков программирования. Рисунок 3.1 изображает переменные, контролирующие трансформацию одного типа файлов в другой.

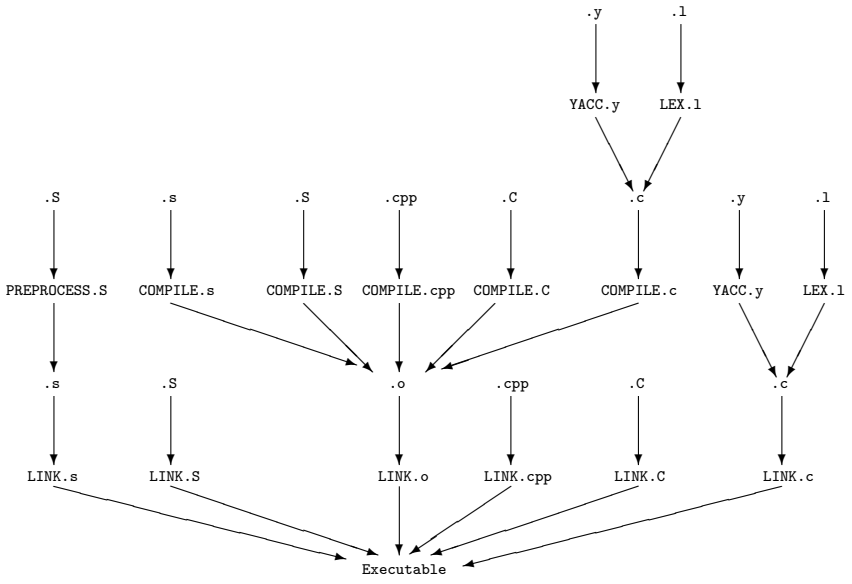

Рис. 3.1: Переменные для компиляции исходных файлов C/C++.

Все эти переменные имеют одинаковую форму: *ДЕЙСТВИЕ.суффикс*. *ДЕЙСТВИЕ* может быть `COMPILE` для создания объектного файла, `LINK` для создания исполняемого файла, или одной из «специальных» операций `PREPROCESS`, `YACC` или `LEX` (для

запуска препроцессора языка C, *yacc* и *lex* соответственно). Тип обрабатываемых файлов указывается с помощью части *суффикс*.

Стандартный «путь» через эти переменные, например, для C++, проходит через два правила. Сначала происходит компиляция исходных файлов C++ в объектные файлы, которые затем компонуются в исполняемый файл.

```
%.o: %.C
    $(COMPILE.C) $(OUTPUT_OPTION) $<

%: %.o
    $(LINK.o) $\ $(LOADLIBES) $(LDLIBS) -o $@
```

Первое правило использует следующие определения переменных:

```
COMPILE.C     = $(COMPILE.cc)
COMPILE.cc    = $(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CXX           = g++
OUTPUT_OPTION = -o $@
```

GNU *make* поддерживает два расширения исходных файлов C++: *.C* и *.cc*. Значением переменной `CXX` является имя компилятора C++, по умолчанию это *g++*. Переменные `CXXFLAGS`, `CPPFLAGS` и `TARGET_ARCH` (опции компилятора C++, опции препроцессора C и опции, специфичные для архитектуры, соответственно) не имеют значения по умолчанию. Они нужны для настройки процесса сборки конечным пользователем. Переменная `OUTPUT_OPTION` содержит имя выходного файла.

Правило компоновки немного проще:

```
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
CC     = gcc
```

Это правило использует компилятор языка C для компоновки объектных файлов в исполняемые. Компилятором языка C по умолчанию является *gcc*. Переменные `LDFLAGS` и `TARGET_ARCH` не имеют значений по умолчанию. Значение `LDFLAGS` содержит опции компоновки, например, флаги `-L`. Переменные `LOADLIBES` и `LDLIBS` содержат список библиотек для компоновки. Такая избыточность была введена из соображений переносимости.

Итак, мы завершили наш краткий обзор переменных *make*. На самом деле их гораздо больше, однако того, что мы рассмотрели, достаточно для того, чтобы понять связь переменных и правил. Например, существует группа переменных, предназначенных для работы с TEX, и набор соответствующих правил. Рекурсивный вызов *make* — ещё одна тема, для обсуждения которой нам понадобятся переменные. Мы вернёмся к ней в главе 6.

Глава 4

Функции

GNU *make* поддерживает как встроенные, так и определяемые пользователем функции. Вызов функций во многом подобен обращению к переменной, с той разницей, что содержит один или более параметров, разделённых запятыми. Результат вычисления встроенных функций обычно присваивается некоторой переменной или передаётся в дочерний процесс командного интерпретатора. Функции, определяемые пользователем, хранятся в виде значения переменной или тела макроса и ожидают получить на вход один или более аргументов.

4.1 Функции, определяемые пользователем

Хранение последовательностей команд в переменных открывает широкие возможности для приложений. Например, вот небольшой макрос для завершения процесса[1]:

```
AWK  := awk
KILL := kill

# $(kill-acroread)

define kill-acroread
  @ ps -W |                                          \
  $(AWK) 'BEGIN      { FIELDWIDTHS = "9 47 100" }     \
          /AcroRd32/ {                               \
                       print "Killing " $$3;         \
                       system( "$(KILL) -f " $$1 )    \
```

[1] «Зачем нам делать это в *makefile*'е?», — спросите вы. Например, в Windows, открытие файла одним процессом делает его недоступным для записи другому процессу. Пока я писал эту книгу, PDF файл часто был заблокирован процессом программы Acrobat Reader, что не допускало обновление PDF-файла с помощью *make*. Поэтому я добавил эту команду в несколько целей, чтобы завершать процесс Acrobat Reader и открывать доступ к заблокированному файлу.

```
                }'
endef
```

Этот макрос был написан с использованием инструментов Cygwin[2], поэтому имена программ и опции команд *ps* и *kill* нестандартны для Unix. Для завершения программы мы направляем вывод программы *ps* на вход *awk*. Сценарий *awk* ищет процесс Acrobat Reader по его имени в системе Windows, и завершает его в случае необходимости. Мы использовали возможности, предоставляемые переменной `FIELDWIDTH`, чтобы трактовать имя команды со всеми аргументами как одно поле с точки зрения *awk*. Поэтому сценарий корректно напечатает полное имя запущенной программы вместе с аргументами, даже если оно содержит пробелы. Ссылки на поля в *awk* обозначаются как `$1`, `$2` и так далее. Если обращения к полям не экранировать, они могут быть рассмотрены как переменные *make*. Мы можем сообщить *make*, что `$n` не нужно вычислять, с помощью экранирования символа доллара в `$n` вторым символом доллара, `$$n`. В этом случае *make* увидит два идущих подряд символа доллара, уберёт лишний и передаст оставшийся в дочерний процесс командного интерпретатора.

Хороший макрос. К тому же, директива `define` предохраняет нас от дублирования кода в случае, если нам придётся использовать макрос часто. И всё же он далёк от совершенства. Что если мы захотим завершить произвольный процесс, не только процесс Acrobat Reader? Придётся ли нам определить ещё один макрос и написать сценарий заново? Нет!

Переменным и макросам можно передавать аргументы, таким образом, каждое их вычисление может отличаться от предыдущего. Параметры макроса доступны в его теле через позиционные переменные `$1`, `$2` и так далее. В качестве параметра нашей функции *kill-acroread* подходит шаблон поиска:

```
AWK        := awk
KILL       := kill
KILL_FLAGS := -f
PS         := ps
PS_FLAGS   := -W
PS_FIELDS  := "9 47 100"

# $(call kill-program,awk-pattern)
define kill-program
  @ $(PS) $(PS_FLAGS) |                                  \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }           \
          /$1/  {                                        \
                  print "Killing " $$3;                  \
```

[2]Инструменты Cygwin — это порт многих стандартных утилит GNU и Linux под Windows. Они включают набор компиляторов, X11R6, *ssh* и даже *inetd*. Порт основывается на библиотеке, реализующей системные вызовы Unix в терминах функций Win32 API. Это настоящий инженерный шедевр, и я настоятельно рекомендую его вам. Загрузите его с сайта *http://www.cygwin.com*.

```
                system( "$(KILL) $(KILL_FLAGS) " $$1 ) \
              }'
endef
```

Мы заменили шаблон поиска для *awk*, `/AcroRd32/`, обращением к параметру, `$1`. Заметим, что существует тонкое различие между параметром макроса `$1` и обращением к полю *awk* `$$1`. Очень важно помнить, какая имена программа нуждается в получении значения переменной. Пока мы улучшали нашу функцию, мы также присвоили ей более подходящее имя и заменили жёстко закодированные значения (зависящие от Cygwin) переменными. Теперь мы имеем довольно переносимый макрос для завершения процесса.

Давайте испытаем его на практике:

```
FOP        := org.apache.fop.apps.Fop
FOP_FLAGS  := -q
FOP_OUTPUT := > /dev/null

%.pdf: %.fo
    $(call kill-program,AcroRd32)
    $(JAVA) $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)
```

Это шаблонное правило завершает процесс Acrobat Reader, если он запущен на выполнение, и конвертирует *fo*-файлы (Formatting Objects) в *pdf*-файл вызовом процессора *fop* (*http://xml.apache.org/fop*). Вычисление макроса или переменной имеет следующий синтаксис:

```
$(call имя-макроса[, аргумент1 . . . ])
```

Встроенная функция *call* вычисляет свой первый аргумент, заменяя в нём все вхождения позиционных аргументов (`$1`, `$2` и так далее) остальными своими аргументами. (На самом деле, она вовсе не «вызывает» макрос в смысле передачи управления, а производит специальное вычисление макроса). На месте шаблона *имя-макроса* может быть любая переменная или макрос (вспомним, что макросы — это просто переменные, в значениях которых допускаются символы новой строки). Макрос или переменная даже не обязательно должны содержать позиционные аргументы (`$n`), однако в этом случае использование функции *call* не имеет особого смысла. Аргументы макроса, следующие за параметром *имя-макроса*, разделяются запятыми.

Заметим, что первый аргумент функции *call* является именем, а не значением переменной (то есть не начинается со знака доллара). Это довольно необычно. Существуют и другие встроенные функции, принимающие на вход имена переменных, например, *origin*. Если вы окружите первый аргумент функции *call* знаком доллара и скобочками, его значение будет вычислено и передано на вход *call*.

Проверка аргументов в функции *call* минимальна. На вход может быть передано любое количество аргументов. Если макрос использует позиционный аргумент `$n`, для которого в обращении к *call* нет фактического параметра, позиционный аргумент заменяется пустой строкой. Если аргументов *call* больше, чем позиционных аргументов макроса, они никогда не будут подставлены в его тело.

Если вы вызываете один макрос из другого, вам стоит быть осторожным, поскольку порой GNU *make* 3.80 ведёт себя довольно странно. Функция *call* во время подстановки определяет свои аргументы как обычные переменные *make*. Таким образом, если один макрос вызывает другой, есть возможность, что аргументы вызывающего макроса будут видны вызываемому макросу:

```
define parent
  echo "вызывающий макрос имеет два аргумента: $1, $2"
  $(call child,$1)
endef

define child
  echo "вызываемый макрос имеет один аргумент: $1"
  echo "однако он также имеет доступ к аргументу "
  echo "вызывающего макроса: $2!"
endef

scoping_issue:
    @$(call parent,one,two)
```

После запуска *make* мы сможем наблюдать проблему с областью видимости:

```
$ make

вызывающий макрос имеет два аргумента: one, two
вызываемый макрос имеет один аргумент: one
однако он также имеет доступ к аргументу
вызывающего макроса: two!
```

В версии 3.81 это было исправлено, так что аргумент `$2` вызываемого макроса вычислится как пустая строка.

Мы проведём ещё много времени, составляя собственные функции, однако нам нужно ещё немного знаний, чтобы делать по-настоящему интересные вещи!

4.2 Встроенные функции

Как только вы начнёте использовать переменные для более сложных манипуляций, нежели хранение констант, вы обнаружите потребность манипулировать их содержимым. У вас есть такая возможность. GNU *make* имеет несколько десятков встроенных функций для работы с переменными и их содержимым. Все функции

попадают в одну из следующих категорий: функции для анализа и подстановки строк; функции для обработки имён файлов; функции управления выполнением; функции, определяемые пользователем и вспомогательные функции.

Но сначала рассмотрим синтаксис обращения к функций. Вызов любой функции имеет следующую форму:

```
$(имя-функции аргумент1 [, аргумент2, ...])
```

Сначала указывается `$(`, затем следует имя функции, за которым следуют аргументы. Начальный пробел у первого аргумента вырезается, остальные аргументы подставляются со всеми начальными (и, конечно, внутренними и оконечными) пробелами. Аргументы функций разделяются запятыми, таким образом, при вызове функции с одним аргументом запятые не требуются, при вызове функции с двумя аргументами используется одна запятая, и так далее. Многие функции принимают один аргумент, интерпретируя его как последовательность слов, разделённых пробелами. Для таких функций пробел считается разделителем слов и попросту игнорируется, если таковым не является.

Мне нравится использовать пробелы. Они делают код более читабельным и удобным для поддержки. Поэтому я буду использовать пробелы везде, где это сойдёт мне с рук. Однако иногда пробелы в списке аргументов или определении переменной могут мешать корректному выполнению кода. Когда это случается, у вас не остаётся другого выбора, кроме как убрать лишний пробел. Мы уже видели один пример, в котором лишний пробел случайно попал в шаблон поиска программы *grep*. Мы будем указывать, где пробелы могут вызвать проблемы, при разборе конкретных примеров.

Многие функции *make* принимают в качестве аргумента шаблон. Этот шаблон следует тому же синтаксису, что и шаблоны, используемые в шаблонных правилах (см. раздел «Шаблонные правила» главы 2). Шаблон может содержать один символ `%` с некоторым суффиксом или префиксом (или и тем, и другим). Символ `%` соответствует нулю или более произвольных символов. Шаблону может соответствовать только вся строка, но не подмножество символов внутри строки. Позже мы проиллюстрируем это коротким примером. Символ `%` опционален и при желании его можно опустить.

4.2.1 Строковые функции

Бóльшая часть функций *make* переводит текст из одной формы в другую, однако в этом разделе мы рассмотрим только те из них, которые предоставляют наиболее мощную функциональность по манипулированию строками.

Одной из наиболее общих операций над строками в *make* является выделение подмножества файлов из списка. В командных сценариях для этого обыч-

но используется *grep*. В *make* мы можем использовать функции *filter*, *filter-out* и *findstring*.

`$(filter` *шаблон...*`,`*текст*`)`

Функция *filter* интерпретирует аргумент *текст* как последовательность слов, разделённых пробелами и возвращает список тех из них, которые соответствуют *шаблону*. Например, для для сборки библиотеки пользовательского интерфейса, мы можем выбрать все объектные файлы из подкаталога *ui*. В следующем примере мы извлечём из списка всех имён фалов проекта имена, начинающиеся с *ui/* и заканчивающиеся *.o*. Символ `%` соответствует любому числу символов между префиксом и суффиксом:

```
$(ui_library): $(filter ui/%.o,$(objects))
    $(AR) $(ARFLAGS) $@ $^
```

Функция *filter* может принимать несколько шаблонов, разделённых пробелами. Как уже упоминалось ранее, для того, чтобы некоторое слово было включено в выходной список, оно должно соответствовать шаблону целиком. Рассмотрим следующий пример:

```
words := he the hen other the%

get-the:
    @echo he matches:   $(filter he,   $(words))
    @echo %he matches:  $(filter %he,  $(words))
    @echo he% matches:  $(filter he%,  $(words))
    @echo %he% matches: $(filter %he%, $(words))
```

Когда мы запустим *make*, вывод будет следующим:

```
$ make

he matches: he
%he matches: he the
he% matches: he hen
%he% matches: the%
```

Как видите, первый шаблону соответствует только слово `he`, так как на выход *filter* попадают только слова, соответствующие шаблону целиком. Остальным шаблонам соответствует слово `he` и слова, содержащие `he` в нужной позиции.

Шаблон может содержать только один метасимвол `%`. Дополнительные символы `%` воспринимаются как литералы.

Может показаться странным, что функция *filter* не может находить подстроки внутри слова или принимать более одного метасимвола. Во многих случаях вам будет не хватать подобной функциональности. Однако вы можете реализовать нечто похожее с помощью циклов и условных операторов. Позже мы рассмотрим, как это можно сделать.

`$(filter-out` *шаблон...,текст*`)`

Функция *filter-out* осуществляет действие, обратное действию функции *filter*, отбирая слова, не соответствующие шаблону. В следующем примере мы отсеим все заголовочные файлы C:

```
all_source := count_words.c counter.c lexer.l \
              counter.h lexer.h
to_compile := $(filter-out %.h, $(all_source))
```

`$(findstring` *строка,текст*`)`

Эта функция ищет вхождение *строки* в *тексте*. Если вхождение обнаружено, функция возвращает *строку*, иначе возвращается пустая строка.

Поначалу может показаться, что эта функция подобна функциональности поиска подстрок программы *grep* (ранее мы выдвигали на эту роль функцию *filter*), но это не так. Первое и самое важное отличие заключается в том, что *findstring* возвращает только искомую подстроку, а не всё слово, в которое эта подстрока входит. Вторым отличием является отсутствие возможности использования метасимвола `%` (если поместить этот символ в *строку*, он будет восприниматься как литерал).

Наиболее часто эта функция используется совместно с функцией *if*, обсуждаемой дальше в этой главе. Однако есть одна ситуация, в которой функция *findstring* полезна сама по себе.

Предположим, что у нас есть несколько деревьев каталогов в файловой системе, имеющих сходную структуру, например, каталоги исходного кода приложения, исходного кода исполняющей среды, бинарных файлов с таблицей символов и бинарных файлов, скомпилированных с флагами оптимизации. Скорее всего, в ваших сценариях вам потребуется узнавать, внутри какого именно дерева вы находитесь (не получая при этом относительный путь от корневого каталога проекта). Вот набросок кода для получения этой информации:

```
find-tree:
    # PWD = $(PWD)
    # $(findstring /test/book/admin,$(PWD))
    # $(findstring /test/book/bin,$(PWD))
```

```
    # $(findstring /test/book/dblite_0.5,$(PWD))
    # $(findstring /test/book/examples,$(PWD))
    # $(findstring /test/book/out,$(PWD))
    # $(findstring /test/book/text,$(PWD))
```

Каждая строка начинается с символа табуляции и символа комментария *shell*, поэтому «выполняется» в отдельном дочернем процессе командного интерпретатора, как и обычные команды. Командный интерпретатор «Bourne Again Shell», *bash*, и многие родственные ему интерпретаторы просто проигнорируют эти строки. Это более распространённый способ распечатки результатов вычисления конструкций *make*, чем использование команды `@echo`. Вы можете добиться того же эффекта, если будете использовать более переносимый оператор командного интерпретатора `:`, однако оператор `:` осуществляет перенаправление потоков. Таким образом, команда, содержащая подстроку `> слово`, в качестве побочного эффекта создаст файл `слово`. После запуска предыдущего примера мы получим следующий вывод:

```
$ make

# PWD = /test/book/out/ch03-findstring-1
#
#
#
#
# /test/book/out
#
```

Как вы можете видеть, каждый тест переменной `$(PWD)` возвращал пустую строку, пока не встретился нужный каталог. Как уже было замечено, этот код лишь демонстрирует возможности *findstring* и может быть использован для определения текущего дерева каталогов.

Существует две функции для поиска и замены строк:

`$(subst` *строка-поиска*`,`*строка-замены*`,`*текст*`)`

Это функция для простого (не содержащего шаблонов) поиска и замены. Одно из наиболее частых применений этой функции — замена суффиксов в списке имён файлов:

```
sources := count_words.c counter.c lexer.c
objects := $(subst .c,.o,$(sources))
```

В предыдущем примере все вхождения строки «*.c*» в значении переменной `$(sources)` будут заменены строкой «*.o*», или, более óбщно, все вхождения *строки-поиска* будут заменены *строкой-замены*.

Эта функция также является примером ситуации, когда пробелы в списке аргументов функции имеют значение. Обратите внимание, что после запятых нет пробелов. Если бы мы написали такой код:

```
sources := count\_words.c counter.c lexer.c
objects := $(subst .c, .o, $(sources))
```

(обратите внимание на пробелы после запятых), то значением переменной `$(objects)` стала бы следующая строка:

```
count_words .o counter .o lexer .o
```

Это не совсем то, что мы хотим. Проблема заключается в том, что пробел перед аргументом `.o` является частью *строки-замены*, и поэтому попадает в результирующий текст. Пробел перед аргументом `.c` не вызовет проблем, потому что *make* пропускает пробелы перед первым аргументом. Пробел перед `$(sources)` также не вызовет проблемы, поскольку переменная `$(objects)` скорее всего будет использоваться как простой аргумент командной строки. Однако я стараюсь никогда не смешивать различные стили расстановки пробелов при вызове функций, даже если это приводит к корректным результатам:

```
# Пожалуй, не самая удачная расстановка пробелов.
objects := $(subst .c,.o, $(source))
```

Заметим, что *subst* не понимает имён файлов или их суффиксов, только последовательности символов. Если какой-то файл с исходным кодом содержит внутри имени подстроку `.c`, то она будет заменена. Например, имя файла *car.cdr.c* будет преобразовано в *car.odr.o*. Вполне вероятно, что это не то, что вы ожидали.

В разделе «Автоматическое определение зависимостей» главы 2 мы обсуждали составление зависимостей. Последний пример *makefile*'а этого раздела использовал *subst* следующим образом:

```
VPATH = src include
CPPFLAGS = -I include
SOURCES = count_words.c \
          lexer.c       \
```

```
          counter.c
count_words: counter.o lexer.o -lfl
count_words.o: counter.h
counter.o: counter.h lexer.h
lexer.o: lexer.h
include $(subst .c,.d,$(SOURCES))

%.d: %.c
    $(CC) -M $(CPPFLAGS) $< > $@.$$$$;                    \
    sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

Функция *subst* использована для преобразования списка файлов с исходным кодом в список файлов зависимостей. Поскольку файлы зависимостей появляются в качестве директивы `include`, они рассматриваются как реквизиты и создаются при помощи шаблонного правила `%.d`.

`$(patsubst` *шаблон-поиска*`,`*шаблон-замены*`,`*текст*`)`

Эта функция отличается от предыдущей тем, что допускает использование шаблонов для поиска и замены символов. Шаблон, как обычно, может содержать один и только один метасимвол `%`. Метасимвол `%` в *строке-замены* заменяется соответствующим текстом. Важно помнить, что *текст* должен полностью соответствовать *шаблону-поиска*. Например, следующий пример удалит только последний (не каждый) слэш в *тексте*:

```
strip-trailing-slash = $(patsubst %/,%,$(directory-path))
```

Использование *подстановочной ссылки (substitution reference)* является переносимым способом осуществления такой замены. Синтаксис использования подстановочной ссылки таков:

`$(`*имя-переменной*`:`*шаблон-поиска*`=`*шаблон-замены*`)`

Шаблон-поиска может быть простой строкой, в этом случае он заменяется на *шаблон-замены* в конце каждого слова, то есть когда за ним следует пробел или конец значения переменной. *Шаблон-поиска* может также содержать символ `%`, в этом случае поиск и замена осуществляется по тем же правилам, что и для функции *patsubst*. Я нахожу этот синтаксис более трудным для чтения и восприятия по сравнению с *patsubst*.

Как мы уже видели, переменные часто состоят из списка слов. Ниже представлены функции для выбора слов из списка, подсчёта длины списка и т.д. Как и в случае остальных функций *make*, слова в списке разделяются пробелами.

`$(words` *список*`)`
Функция, возвращающая число слов в *списке*.

```
CURRENT_PATH := $(subst /, ,$(HOME))
words:
    @echo Мой домашний каталог содержит \
$(words $(CURRENT_PATH)) подкаталогов.
```

Эта функция имеет множество применений, однако для того, чтобы эффективно её использовать, нам нужно изучить ещё несколько функций.

`$(word` *n*`,`*список*`)`
Эта функция возвращает слово, находящееся в *списке* под номером *n*, первое слово списка имеет номер 1. Если *n* больше числа слов в *тексте*, функция возвращает пустую строку.

```
version_list  := $(subst ., ,$(MAKE_VERSION))
minor_version := $(word 2, $(version_list))
```

Переменная `MAKE_VERSION` является стандартной переменной (см. раздел «Стандартные переменные *make*» главы 3). Вы можете получить последнее слово в списке следующим образом:

```
current := $(word $(words $(MAKEFILE_LIST)),\
                          $(MAKEFILE_LIST))
```

Такая конструкция вернёт последний прочитанный *makefile*.

`$(firstword` *список*`)`
Эта функция возвращает первое слово в *списке*. Её действие эквивалентно вызову `$(word 1,`*список*`)`.

```
version_list  := $(subst ., ,$(MAKE_VERSION))
major_version := $(firstword $(version_list))
```

`$(lastword` *список*`)`
Эта функция, доступная в версиях GNU *make* 3.81 и выше, представляет дополнение к функции *firstword*. Она возвращает последнее слово в *списке*, что функционально эквивалентно следующему выражению:

```
$(word $(words список),список)
```

`$(wordlist` *номер-начала*`,`*номер-конца*`,`*список*`)`

Эта функция возвращает слова, имеющие в *списке* номера с *номер-начала* по *номер-конца* включительно. Как и в функции *word*, первое слово имеет номер 1. Если *номер-начала* больше числа слов в *списке*, то функция вернёт пустую строку. Если *номер-конца* больше числа слов в *списке*, функция вернёт всё слова начиная с *номера-начала*.

```
# $(call uid_gid, user-name)
uid_gid = $(wordlist 3, 4,   \
            $(subst :, ,     \
              $(shell grep "^$1:" /etc/passwd)))
```

4.2.2 Некоторые важные функции

Перед тем, как мы рассмотрим функции для работы с именами файлов, следует ввести две чрезвычайно полезные функции: *sort* и *shell*.

`$(sort` *список*`)`

Эта функция сортирует слова в *списке*, удаляя дубликаты. Результирующий список содержит уникальные слова из *списка* в лексикографическом порядке, разделённые одним пробелом. В дополнение функция *sort* удаляет начальные и конечные пробелы.

```
$ make -f- <<< 'x:;@echo =$(sort d b s d t )='
=b d s t=
```

Функция *sort*, разумеется, реализована непосредственно *make* и потому не поддерживает опций стандартной утилиты *sort*. Эта функция оперирует лишь своим аргументом, обычно это переменная или результат выполнения другой функции.

`$(shell` *команда*`)`

Функция *shell* принимает на вход один аргумент, который подвергается вычислению (как аргументы всех функций) и передаётся в дочерний процесс командного интерпретатора для выполнения. стандартный вывод команды считывается и возвращается в качестве значения функции. Последовательности символов новой строки заменяются одним пробелом, начальные и конечные символы новой строки удаляются. Данные из стандартного потока ошибок не возвращаются, так же как и код возврата команды.

```
stdout := $(shell echo normal message)
stderr := $(shell echo error message 1>&2)
shell-value:
    # $(stdout)
    # $(stderr)
```

Как вы можете видеть, сообщения из стандартного потока ошибок попадают на терминал, и потому не включаются в вывод функции *shell*:

```
$ make
error message
# normal message
#
```

Ниже приведён пример использования этой функции для создания набора каталогов:

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS);  \
             do                                \
                 [[ -d $$d ]] || mkdir -p $$d; \
             done)
```

Часто *makefile* можно сделать проще, если все каталоги, в которые *make* осуществляет запись файлов, гарантированно существуют до выполнения первой команды. Предыдущая переменная содержит код, создающий требуемые каталоги при помощи цикла `for` командного интерпретатора *bash*. Двойные квадратные скобки являются стандартным синтаксисом *bash*, они работают почти также, как программа *test*. Отличие заключается в том, что внутри двойных квадратных скобок не осуществляется разделение на слова и подстановка путей. Поэтому если имя файла содержит пробел, тест будет выполнен успешно (причём имя переменной не придётся заключать в кавычки). Поскольку мы поместили определение переменной `_MKDIRS` в начале *makefile*'а, мы можем быть уверены в том, что её вычисление произойдёт до того, как другие команды или вычисления переменных обратятся к каталогам. Значение `_MKDIRS` для нас не важно и никогда не будет использоваться.

Поскольку функция *shell* может быть использована для запуска произвольной внешней программы, вам следует использовать её осторожно. В частности, вам следует учитывать различия между простыми и рекурсивными переменными.

```
START_TIME  := $(shell date)
CURRENT_TIME = $(shell date)
```

Определение переменной `START_TIME` приводит к выполнению программы *date*. Переменная `CURRENT_DATE` будет выполнять *date* каждый раз при необходимости использования значения в *makefile*'е.

Теперь у нас достаточно инструментов для того, чтобы писать довольно интересные функции. Ниже приведён пример функции, которая проверяет, содержит ли её аргумент одинаковые слова.

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter                \
                   $(words $1),          \
                   $(words $(sort $1)))
```

Мы считаем слова в первоначальном списке и в списке, из которого были удалены все дубликаты, а затем «сравниваем» эти два числа. *make* не содержит функций, принимающий числа в качестве аргументов, все аргументы воспринимаются как строки. Поэтому для того, чтобы сравнить два числа, мы должны сравнить их как строки. Наиболее простым способом сделать это является применение функции *filter*. Мы «ищем» одно число в другом. Функция *has-duplicates* вернёт непустую строку только в том случае, если её аргумент содержит одинаковые слово.

А вот простой способ создать файл с временно́й меткой в имени:

```
RELEASE_TAR := mpwm-$(shell date +%F).tar.gz
```

Это выражение порождает следующую строку:

```
mpwm-2003-11-11.tar.gz
```

Мы могли бы достигнуть того же эффекта, заставив *date* проделать немного больше работы:

```
RELEASE_TAR := $(shell date +mpwm-%F.tar.gz)
```

Следующая функция может быть использована для получения полного имени Java-класса из относительного пути (возможно, начиная с каталога *com*).

```
\# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(patsubst %.java,%,$1))
```

Этот же эффект может быть достигнут при помощи следующего кода:

```
# $(call file-to-class-name, file-name)
file-to-class-name := $(subst /,.,$(subst .java,,$1))
```

Мы можем использовать предыдущий пример для вызова классов Java:

```
CALIBRATE_ELEVATOR := com/wonka/CalibrateElevator.java

calibrate:
    $(JAVA) $(call file-to-class-name,$(CALIBRATE_ELEVATOR))
```

Если переменная `$(sources)` содержит имена каталогов, содержащих каталог *com*, их можно убрать при помощи следующей функции (корень дерева каталогов должен быть передан в качестве первого аргумента[3]):

```
# $(call file-to-class-name, root-dir, file-name)
file-to-class-name := $(subst /,.,                \
                        $(subst .java,,           \
                          $(subst $1/,,$2)))
```

Функции, подобные этой, обычно легче понять, если читать их изнутри. Начиная с внутреннего вызова *subst*, функция удаляет строку `$1/`, затем удаляет строку `.java` и, наконец, заменяет все слэши точками.

4.2.3 Функции для работы с файлами

Создатели *makefile*'ов тратят значительную часть времени на осуществление обработки файлов. Поэтому неудивительно, что *make* содержит так много функций для облегчения этой задачи.

`$(wildcard` *шаблон* `...)`

Шаблоны были рассмотрены в главе 2 в контексте целей, реквизитов и командных сценариев. Но что, если мы хотим получить эту функциональность в другом контексте, например, в контексте определения переменной? Конечно, мы могли бы использовать функцию *shell* для подстановки шаблона с помощью дочернего процесса командного интерпретатора, однако это будет очень неэффективным решением, если подобная операция будет осуществляться достаточно часто. Альтернативным подходом является использование функции *wildcard*:

```
sources := $(wildcard *.c *.h)
```

Функция *wildcard* принимает список шаблонов в качестве аргумента и производит поиск файлов, соответствующих хотя бы одному шаблону[4]. Если не обнаруживается файлов, соответствующих хотя-бы одному шаблону, функция возвращает пустую строку. Как и в случае целей и реквизитов, в шаблонах могут использоваться стандартные метасимволы `~`, `*`, `?`, `[...]`, `[^ ...]`.

[3] Существуют соглашения относительно структуры каталогов исходного кода Java: все классы должны быть объявлены в пакете, содержащем доменное имя разработчика, записанное в обратном порядке (сначала следуют домены более высокого уровня), а структура каталогов отражает структуру пакетов. Однако довольно часто можно встретить подобную структуру каталогов: *root-dir/com/company-name/dir* (прим. автора).

[4] Руководство пользователя GNU *make* 3.80 почему-то умалчивает о возможности использования нескольких шаблонов (прим. автора).

Другим применением функции *wildcard* является проверка существования файлов в условных выражениях. Довольно часто встречается использование *wildcard* с аргументом, не содержащим метасимволов, совместно с функцией *if*. Например, в следующем отрывке переменная `dot-emacs-exists` будет содержать непустое значение только в том случае, если домашний каталог текущего пользователя содержит файл *.emacs*:

```
dot-emacs-exists := $(wildcard ~/.emacs)
```

`$(dir` *список* `...)`

Функция *dir* возвращает каталог для каждого файла из *списка*. Следующее выражение вернёт все каталоги, содержащие исходные файлы C:

```
source-dirs = $(sort                              \
                $(dir                             \
                  $(shell find . -name '*.c')))
```

Программа *find* вернёт список всех файлов с исходным кодом, функция *dirs* отсечёт имя файла, оставив лишь название соответствующего каталога, а функция *sort* удалит повторяющиеся каталоги. Заметим, что переменная `source-dirs` объявлена как простая для избежания повторного вызова *find* при каждом обращении к ней (предполагается, что файлы с исходным кодом не будут спонтанно появляться и исчезать во время выполнения *makefile*'а). Ниже приведён пример функции, требующей рекурсивной переменной:

```
# $(call source-dirs, dir-list)
source-dirs = $(sort                              \
                $(dir                             \
                  $(shell find $1 -name '*.c')))
```

Эта версия принимает в качестве аргумента список каталогов для поиска с пробелом в качестве разделителя. Первым аргументом программы *find* является список каталогов для поиска. Окончания списка каталогов распознаётся благодаря символу тире, содержащемуся в названии следующего аргумента[5].

`$(notdir` *список* `...)`

Функция *notdir* возвращает имена файлов для заданного списка путей. Ниже приведён пример вычисления имени Java-класса по названию файла с его исходным кодом:

[5]Это одна из возможностей *find*, о которых я не знал в течении десятилетий! (прим. автора)

```
# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(subst .java,,$1))
```

Существует много примеров совместного применения функций *dir* и *notdir* для получения желаемого результата. Предположим, что некоторый командный сценарий должен выполняться в том же каталоге, что и файл, создаваемый этим сценарием.

```
$(OUT)/myfile.out: $(SRC)/source1.in $(SRC)/source2.in
    cd $(dir $@); \
    generate-myfile $^ > $(notdir $@)
```

Автоматическая переменная `$@`, представляющая имя цели, может подвергаться декомпозиции для получения имени каталога цели и названия файла цели по-отдельности. Если `OUT` содержит абсолютный путь, вовсе не обязательно использовать функцию *notdir*, однако её применение сделает вывод *make* более удобным для чтения.

Ещё одним способом декомпозиции имени файла является использование переменных `$(@D)` и `$(@F)`, рассмотренных в разделе «Автоматические переменные» главы 2.

`$(suffix` ***список*** `...)`

Функция *suffix* возвращает суффикс (расширение) каждого файла в ***списке***. Ниже приведён пример функции, проверяющей, все ли слова в списке имеют одинаковый суффикс:

```
# $(call same-suffix, file-list)
same-suffix = $(filter 1,$(words $(sort $(suffix $1))))
```

Наиболее часто функция *suffix* применяется в условных выражениях совместно с функцией *findstring*.

`$(basename` ***список*** `...)`

Функция *basename* является дополнением к *suffix*. Она возвращает имя файла без суффикса, оставляя нетронутыми любые начальные компоненты имени файла. Вот пример альтернативной реализации функций *file-to-class* и *get-java-class-name* с использованием функции *basename*:

```
# $(call file-to-class-name, root-directory, file-name)
file-to-class-name := $(subst /,.,           \
                        $(basename           \
                          $(subst $1/,,$2)))

# $(call get-java-class-name, file-name)
get-java-class-name = $(notdir $(basename $1))
```

`$(addsuffix` *суффикс*`,`*список* `...)`

Функция *addsuffix* добавляет заданный *суффикс* к имени каждого файла в *списке*. В качестве *суффикса* может выступать произвольная строка. Ниже приведён пример функции для поиска всех файлов из `PATH`, соответствующих заданному шаблону:

```
# $(call find-program, filter-pattern)
find-program = \
    $(filter $1,                    \
      $(wildcard                    \
        $(addsuffix /*,             \
          $(sort                    \
            $(subst :, ,            \
              $(subst ::,:.:,       \
                $(patsubst :%,.:%, \
                  $(patsubst %:,%:.,$(PATH)))))))))

find:
    @echo $(words $(call find-program, %))
```

Три внутренних подстановки осуществляются для корректной работы в особых случаях. Пустой компонент `PATH` может быть использован для обозначения текущего каталога. Для нормализации этого синтаксиса мы ищем пустые компоненты `PATH` в конце строки, в начале строки и внутри строки (именно в таком порядке). Все найденные компоненты заменяются символом «.». Затем разделитель компонент `PATH` заменяется пробелом. Функция *sort* используется для удаления повторяющихся компонентов. После этого к каждому слову в полученном списке добавляется суффикс `/*`, затем вызывается функция *wildcard*, осуществляющая поиск всех файлов, соответствующих шаблону. Наконец, функция *filter* отбирает только те файлы, которые соответствуют заданному шаблону.

Хотя может показаться, что эта функция будет выполняться очень медленно (и может быть действительно так на многих системах), на моём 1.9GHz P4 с 512MB оперативной памяти эта функция выполняется за 0,20 секунды и находит 4335 программ. Можно повысить производительность, переместив аргумент `$1` внутрь вызова *wildcard*. Следующая версия устраняет необходимость вызова *filter* и вызывает *addsuffix* с аргументом, переданным вызывающей функцией:

```
# $(call find-program,wildcard-pattern)
find-program =
    $(wildcard                  \
      $(sort                    \
        $(addsuffix /$1,        \
```

```
            $(subst :, ,           \
              $(subst ::,:.:,      \
                $(patsubst :%,.:%, \
                  $(patsubst %:,%:.,$(PATH))))))))

find:
    @echo $(words $(call find-program,*))
```

Этот вариант выполняется за 0,17 секунды. Он работает быстрее потому, что функция *wildcard* больше не возвращает все файлы каталогов только для того, чтобы затем отсеить их с помощью *filter*. Подобный пример можно встретить в руководстве пользователя GNU *make*. Заметим, что первая версия использует шаблоны в стиле *filter* (содержащие только метасимвол `%`), а вторая — шаблоны в стиле *wildcard* (`~`, `*`, `?`, `[...]` и `[^ ...]`).

`$(addprefix` *`префикс`*`,`*`список`* `...)`

Функция *addprefix* является дополнением к функции *addsuffix*. Ниже приведён пример функции, проверяющей набор файлов на существование и ненулевую длину:

```
# $(call valid-files, file-list)
valid-files = test -s . $(addprefix -a -s ,$1)
```

Функция *valid-files* отличается от большинства рассмотренных тем, что её значение должно быть выполнено в командном интерпретаторе. Для осуществления проверки используется программа *test* с ключом `-s` (истина, если файл существует и не пуст). Поскольку для корректной работы программы *test* имена файлов должны быть разделены ключом `-a`, мы используем *addprefix* для добавления этого ключа к каждому имени файла в списке. В начале цепочки используется каталог ., для него условие существования и непустоты всегда является истинным.

`$(join` *`список-префиксов`*`,`*`список-суффиксов`*`)`

Функция *join* является дополнением к функциям *dir* и *notdir*. Она принимает в качестве аргументов два списка и производит конкатенацию первого элемента из *`списка-префиксов`* с первым элементом из *`списка-суффиксов`*, второго элемента из *`списка-префиксов`* со вторым элементом из *`списка-суффиксов`* и так далее. Эта функция может использоваться для реконструкции списков, декомпозированных при помощи функций *dir* и *notdir*.

`$(abspath` *`список`*`...)`

Для каждого файла из *`списка`* возвращает абсолютный путь, не содержащий компонентов . и .., а также повторяющихся разделителей (/). Заметим, что в отличие от функции *realpath*, *abspath* не производит разрешения

символических ссылок и не требует существования файлов. Для проверки существования файлов используйте функцию *wildcard*.

Эта функция доступна в версиях GNU *make* 3.81 и выше.

`$(realpath` *список*`...)`

Функция *realpath* подобна функции *abspath* с тем отличием, что осуществляет разрешение символических ссылок и проверку существования файлов и каталогов.

Эта функция доступна в версиях GNU *make* 3.81 и выше.

4.2.4 Функции управления выполнением

Поскольку многие из рассмотренных нами функций разработаны для выполнения операций над списками, они прекрасно выполняют свою работу без необходимости применения циклов. Однако без настоящих операторов цикла и условного выполнения макроязык *make* был бы очень ограничен. К счастью, *make* предоставляет возможность использовать оба этих оператора. Я также добавил в этот раздел функцию *error*, являющуюся довольно экстремальной формой управления выполнением.

`$(if` *условие*`,`*then-часть*`,`*else-часть*`)`

Функция *if* (не путайте с директивами условной обработки `ifeq`, `ifneq`, `ifdef` и `ifndef`, рассмотренными в главе 3) выбирает одну из двух подстановок в зависимости от «значения» условного выражения. Значение *условия* считается истинным, если оно содержит хотя бы один символ (например, пробел). В этом случае подставляется *then-часть*. В противном случае, если значение *условия* является пустой строкой, подставляется *else-часть*[6].

Ниже приведёт простой способ узнать, исполняется ли ваш *makefile* из-под Windows. Проверим, есть ли в окружении переменная `COMSPEC`, определённая только в Windows:

```
PATH_SEP := $(if $(COMSPEC),;,:)
```

make проверяет *условие*, предварительно убрав начальные и конечные пробелы и вычислив выражение. Если результатом вычисления будет непустая

[6] В главе 3 мы обсуждали разницу между макроязыками и обычными языками программирования. Макроязыки трансформируют исходный код в текст с помощью определения и подстановки макросов. Это различие станет яснее, когда мы увидим, как работает функция *if*. (прим. автора)

строка, *условие* считается истинным. В итоге PATH_SEP содержит правильный разделитель путей в переменной PATH, независимо от того, выполняется ли ваш *makefile* в Windows или Unix.

В одной из предыдущих глав мы упоминали возможность проверки версии *make* с помощью последних возможностей *make* (например, *eval*). Функции *if* и *filter* также часто используются вместе для проверки значения строк:

```
$(if $(filter $(MAKE_VERSION),3.80),,\
  $(error Этот makefile требует GNU make версии 3.80.))
```

Теперь по мере выхода новых версий GNU *make*, выражение может быть расширено для поддержки большего числа версий.

```
$(if $(filter $(MAKE_VERSION),3.80 3.81 3.90 3.92),,\
  $(error Этот makefile требует одну из следующих версий \
    GNU make ....))
```

Эта техника имеет свои недостатки: придётся изменять код после каждой установки свежей версии *make*. С другой стороны, это происходит не так уж часто (например, версия 3.80 была выпущена в октябре 2002 года, а версию 3.81 пришлось ждать вплоть до апреля 2006 года). Предыдущий тест может быть помещён в самое начало *makefile*'а, поскольку *if* либо вычислится как пустая строка, либо вызовет функцию *error*, завершив выполнение *make*.

`$(error` *сообщение*`)`

Функция *error* предназначена для вывода сообщений о фатальных ошибках. После того, как эта функция напечатает *сообщение*, *make* завершит свою работу с кодом возврата 2. Вывод предваряется указанием имени текущего *makefile*'а и номером текущей строки. Ниже приведена реализация общей конструкции программирования *assert* для *make*:

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef

# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef

# $(call assert-not-null,make-variable)
define assert-not-null
```

```
  $(call assert,$($1),The variable "$1" is null)
endef

error-exit:
    $(call assert-not-null,NON_EXISTENT)
```

Первая функция, *assert*, просто проверяет свой первый аргумент на пустоту и выводит пользовательское сообщение об ошибке, если тестируемое значение является пустой строкой. Вторая функция основывается на первой и проверяет с помощью функции *wildcard* существование файла. Отметим, что аргумент может содержать любое число шаблонов.

Третья функция очень полезна и основывается на *вычисляемых именах переменных*. Переменная *make* может содержать что угодно, даже имя другой переменной *make*. Однако если одна переменная содержит имя второй переменной, как вы сможете получить доступ к значению второй переменной? Можно просто вычислить переменную дважды:

```
NO_SPACE_MSG := No space left on device.
NO_FILE_MSG  := File not found.
...
STATUS_MSG   := NO_SPACE_MSG

$(error $($(STATUS_MSG)))
```

Пример выглядит немного надуманным, однако он достаточно прост и прекрасно иллюстрирует основную идею. Переменная `STATUS_MSG` может принимать одно из нескольких сообщений об ошибках путём сохранения имени переменной, содержащей это сообщение. Когда приходит время вывести сообщение об ошибке, сначала вычисляется `STATUS_MSG` для доступа к имени переменной, содержащей сообщение об ошибке, `$(STATUS_MSG)`, затем полученное имя подвергается повторному вычислению для получения текста сообщения, `$($(STATUS_MSG))`. В нашей функции *assert-not-null* мы предполагали, что аргументом функции является имя переменной. После первого вычисления аргумента, `$1`, мы получаем имя переменной, которое сразу же подвергаем повторному вычислению, `$($1)`, для определения значения этой переменной. Если значение не определено, мы можем использовать имя переменной, `$1`, для вывода сообщения об ошибке:

```
$ make
Makefile:14: *** Assertion failed: The variable \
"NON_EXISTENT" is null. Stop.
```

Существует также функция *warning* (см. раздел «Различные вспомогательные функции» далее в этой главе), которая выводит сообщение в том же формате, что и функция *error*, но не прерывает выполнения *make*.

`$(foreach` *переменная*`,`*список*`,`*тело*`)`

Функция *foreach* предоставляет возможность вычислять выражения в цикле. Заметим, что это отличается от повторного вычисления функции от разных аргументов (хотя это также может быть осуществлено с помощью этой функции). Например:

```
letters := $(foreach letter,a b c d,$(letter))
show-words:
    # letters has $(words $(letters)) words: '$(letters)'

$ make
# letters has 4 words: 'a b c d'
```

Когда выполняется этот цикл, переменной цикла `letter` последовательно присваиваются значения *a b c d*, а тело цикла, `$(letter)`, вычисляется один раз для каждого значения. Вычисленный текст аккумулируется в строке, каждое значение тела отделяется от предыдущего пробелами.

Ниже приведена функция для проверки того, был ли определен набор переменных:

```
VARIABLE_LIST := SOURCES OBJECTS HOME
$(foreach i,$(VARIABLE_LIST), \
  $(if $($i),,                \
    $(shell echo $i has no value > /dev/stderr)))
```

Использование псевдофайла */dev/stderr* требует, чтобы переменная окружения `SHELL` имела значение *bash*. Этот цикл присваивает переменной `i` поочерёдно каждое слово из переменной `VARIABLE_LIST`. Условное выражение внутри *if* сначала вычисляет `$i` для получения имени переменной, а затем находит значение `$($i)` для проверки значения этой переменной на пустоту. Если вычисленная строка не пуста, *then-часть* ничего не делает; в противном случае *else-часть* выводит предупреждение. Заметим, что если мы не перенаправим вывод *echo*, вывод команды *shell* будет подставлен в *makefile*, вызвав синтаксическую ошибку. Результатом вычисления цикла *foreach* является пустая строка.

Как и было обещано, ниже представлена функция для соединения в список всех слов, содержащих заданную подстроку:

```
# $(call grep-string, search-string, word-list)
define grep-string
  $(strip                          \
    $(foreach w, $2,               \
      $(if $(findstring $1, $w),   \
        $w)))
endef

words := count_words.c counter.c lexer.l lexer.h counter.h

find-words:
    @echo $(call grep-string,un,$(words))
```

К сожалению, эта функция не понимает шаблоны, однако нахождение простых подстрок вполне ей под силу:

```
$ make
count_words.c counter.c counter.h
```

`$(and` *условие*$_1$`[,...`*условие*$_n$`])`

Функция *and* представляет операцию `AND`. Она производит вычисление всех своих аргументов по порядку. Если результатом вычисления одного из аргументов является пустая строка, процесс вычисления останавливается и функция возвращает пустую строку. Если ни один из аргументов не порождает пустую строку, возвращается результат вычисления последнего аргумента.

Эта функция доступна в GNU *make* версии 3.81 и выше.

`$(or` *условие*$_1$`[,...`*условие*$_n$`])`

Функция *or* представляет операцию `OR`. Она производит вычисление своих аргументов по порядку. Если результатом вычисления хотя бы одного из аргументов является непустая строка, это строка возвращается в качестве значения функции и дальнейшее вычисление не производится. Иначе возвращается пустая строка.

Эта функция доступна в GNU *make* версии 3.81 и выше.

Замечание о скобках в переменных

Как у же было замечено, если имя переменной *make* состоит из одного символа, то его не обязательно заключать в скобки. Например, все основные автоматические переменные состоят из одного символа. Написание автоматических переменных без скобок встречается повсеместно, даже в руководстве по GNU *make*. Однако практически все остальные переменные, даже состоящие из одного символа, в руководстве по GNU *make* заключаются в скобки. Это подчёркивает особую природу

переменных *make*, поскольку практически во всех языках программирования, в которых для работы с переменными используется символ доллара (таких как командные интерпретаторы, *perl*, *awk* и *yacc*), применение скобок не требуется. Одна из самых распространённых ошибок программирования на *make* — пропущенные скобки. Вот пример использования функции *foreach*, содержащий ошибку:

```
INCLUDE_DIRS := ...
INCLUDES     := $(foreach i,$INCLUDE_DIRS,-I $i)
# INCLUDES теперь имеет значение "-I NCLUDE_DIRS"
```

Однако я нахожу, что читать макросы гораздо легче, если использовать переменные из одного символа и опускать ненужные скобки. Например, я считаю, что функцию *has-duplicates* легче прочитать, если не использовать лишние скобки:

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter          \
                   $(words $1)     \
                     $(words $(sort $1))))
```

Сравните с таким вариантом:

```
# $(call has-duplicates, word-list)
has-duplicates = $(filter          \
                   $(words $(1))   \
                     $(words $(sort $(1)))))
```

Однако функция *kill-program* лишь выигрывает от применения дополнительных скобок, поскольку они помогают отличить переменные *make* от переменных командного интерпретатора или переменных других программ:

```
PS        := ps
PS_FIELDS := "9 47 100"
PS_FLAGS  := -W
define kill-program
  @$(PS) $(PS_FLAGS) |                              \
  $(AWK) 'BEGIN { FIELDWIDTHS = $(PS_FIELDS) }      \
          /$(1)/{                                   \
              print "Killing " $$3;                 \
              system( "$(KILL) $(KILLFLAGS) " $$1 ) \
          }'
endef
```

Строка поиска содержит первый аргумент макроса, `$(1)`. Выражения `$$3` и `$$1` относятся к переменным *awk*.

Я опускаю скобки только в том случае, если это сделает код более удобным для чтения. Обычно это касается параметров макросов и переменных цикла *foreach*.

Вы должны использовать тот стиль, который наиболее подходит в вашей ситуации. Если у вас есть малейшие сомнения относительно удобства сопровождения вашего *makefile*'а, следуйте руководству пользователя GNU *make* и заключайте переменные в круглые скобки. Помните, программа *make* создана для решения проблем, связанных с поддержкой программного обеспечения. Если вы будете иметь это в виду во время написания своих *makefile*'ов, вы сможете избежать многих неприятностей.

4.2.5 Различные вспомогательные функции

Наконец, мы рассмотрим несколько полезных (но не очень важных) функций для манипулирования строками. Скорее всего, вы будете использовать их довольно часто, но не так часто, как *foreach* или *call*.

`$(strip` *`текст`*`)`

Функция *strip* удаляет из *текста* все начальные и конечные пробелы, а также заменяет все повторяющиеся внутренние пробелы одним. Наиболее часто эта функция используется для обработки переменных, использующихся в условных выражениях.

Я часто использую эту функцию для удаления нежелательных пробелов из определений переменных и макросов. Я использовал многострочное форматирование, однако использование функции *strip* для обёртки аргументов `$1`, `$2` и т.д. является неплохой идеей, особенно если вызываемая функция чувствительна к начальным пробелам. Часто программисты, не знающие тонкостей *make*, добавляют пробел после запятой в списке аргументов функции *call*.

`$(origin` *`имя-переменной`*`)`

Функция *origin* возвращает строку, описывающую происхождение переменной. Это может быть очень полезно, чтобы принять решение о том, как использовать значение переменной. Например, вы можете захотеть проигнорировать значение переменной, если она получена из окружения, и использовать её, если она была определена в командной строке. В качестве конкретного примера приведём функцию *assert-defined*, определяющую, определена ли переменная:

```
# $(call assert-defined,variable-name)
define assert-defined
  $(call assert,                          \
    $(filter-out undefined,$(origin $1)), \
    Переменная '$1' не определена)
endef
```

Функция *origin* может возвращать одно из следующих значений:

- `undefined`
  Переменная никогда не была определена.
- `default`
  Переменная определена в стандартной базе данных *make*. Если вы измените значение такой переменной, *origin* вернёт происхождение самого последнего определения.
- `environment`
  Переменная была определена в окружнии (и опция `--environment-overrides` не была включена).
- `environment override`
  Переменная была определена в окружнии (и опция `--environment-overrides` была включена).
- `file`
  Переменная была определена в *makefile*'е.
- `file`
  Переменная была определена в командной строке.
- `override`
  Для определения переменной была использована директива `override`.
- `automatic`
  Переменная является автоматической переменной *make*.

`$(warning` ***сообщение***`)`

Функция *warning* похожа на функцию *error*, но не вызывает завершение выполнения *make*. Вывод ***сообщения*** предваряется именем текущего *makefile*'а и номером текущей строки. Функция *warning* возвращает пустую строку, и потому может использоваться практически везде:

```
$(if $(wildcard $(JAVAC)),,                          \
  $(warning Переменная, определяющая компилятор java, \
            JAVAC ($(JAVAC)), не определена должным   \
            образом.))
```

`$(info` ***сообщение***`)`

Функция *info* выводит ***сообщение*** на стандартный поток вывода, не предваряя его именем *makefile*'а или номером строки. Результатом вычисления этой функции является пустая строка.

Эта функция доступна в GNU *make* версии 3.81 и выше.

`$(value` *`имя-переменной`*`)`

Функция *value* предоставляет способ использовать значение переменной, *не вычисляя* его. Заметим, что откат уже произведённых вычислений не может быть осуществлён. Например, если вы определяете простую переменную (не рекурсивную), её значение вычисляется в момент определения. В этом случае функция *value* вернёт тот же результат, что и обычное вычисление переменной.

`$(flavor` *`имя-переменной`*`)`

Функция *flavor*, в отличие от многих других функций, не оперирует значением переменной, а возвращает некоторую информацию о переменной. Значением этой функции является строка, определяющая тип переменной, имя которой передано в качестве аргумента:

- `undefined`
  Переменная не определена.
- `recursive`
  Переменная является рекурсивно-вычисляемой.
- `simple`
  Переменная является простой.

Эта функция доступна в GNU *make* версии 3.81 и выше.

4.3 Специальные функции

Мы потратим много времени на написание макросов-функций. К сожалению, *make* имеет довольно скудные возможности для их отладки. Давайте попробуем написать простую функцию для вывода отладочной информации.

Как уже было замечено, функция *call* связывает свои параметры с позиционными параметрами `$1`, `$2` и т.д. На вход этой функции может быть передано произвольное число переменных. Особым случаем является имя вызываемой функции (т.е. имя переменной), доступ к которому может быть осуществлён с помощью переменной `$0`. Используя эту информацию, мы можем написан пару отладочных функций для вывода результатов подстановки макросов:

```
# $(debug-enter)
debug-enter = \
    $(if $(debug_trace),\
      $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = \
```

```
    $(if $(debug_trace),$(warning Leaving $0))

comma := ,

echo-args   = \
    $(subst ' ',$(comma) ',\
      $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

Если мы хотим узнать, как вызываются функции *a* и *b*, мы можем использовать описанные выше функции следующим образом:

```
debug_trace = 1

define a
  $(debug-enter)
  @echo $1 $2 $3
  $(debug-leave)
endef

define b
  $(debug-enter)
  $(call a,$1,$2,hi)
  $(debug-leave)
endef

trace-macro:
    $(call b,5,$(MAKE))
```

Разместив переменные *debug-enter* и *debug-leave* в начале и конце ваших собственных функций, вы сможете получать отладочную информацию о их вызове. Однако эти функции далеки от совершенства. Функция *echo-args* выводит только первые девять аргументов, хуже того, она не может определить фактическое число переданных аргументов (конечно, ведь *make* тоже не может этого сделать!). Тем не менее, я использовал эти макросы «как есть» при отладке собственных *makefile*'ов. После выполнения предыдущего примера мы сможем увидеть следующий вывод:

```
$ make
makefile:14: Entering b( '5', 'make', '', ...)
makefile:14: Entering a( '5', 'make', 'hi', '', ...)
makefile:14: Leaving a
makefile:14: Leaving b
5 make hi
```

Как сказал недавно один мой друг, «я никогда раньше не думал о make как о языке программирования». GNU *make* — это не тот *make*, которым пользовались наши прадеды.

4.3.1 Функции *eval* и *value*

Функция *eval* принципиально отличается от остальных встроенных функций. Её назначением является передача текста синтаксическому аранизатору *make*. Вот пример её использования:

```
$(eval sources := foo.c bar.c)
```

Аргумент функции *eval* сначала проверяется на наличие переменных и при необходимости осуществляется подстановка (тоже самое осуществляется для аргументов всех функций), затем полученный текст разбирается и выполняется так, будто он был написан в файле спецификации. Приведённый пример настолько прост, что вы могли удивиться, зачем вам вообще нужна эта функция. Однако давайте рассмотрим более сложный пример. Предположим, у вас есть один *makefile* для сборки десятка различных программ, и вы хотите определить несколько переменных для каждой из этих программ, например, `sources`, `headers` и `objects`. Вместо повторения определений этих переменных снова и снова:

```
ls_sources := ls.c glob.c
ls_headers := ls.h glob.h
ls_objects := ls.o glob.o
...
```

мы можем попробовать определить макрос для выполнения рутинной работы:

```
# $(call program-variables, variable-prefix, file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))
endef

$(call program-variables, ls, ls.c ls.h glob.c glob.h)
show-variables:
    # $(ls_sources)
    # $(ls_headers)
    # $(ls_objects)
```

Макрос *program-variables* принимает два аргумента: префикс для имени трёх переменных и список файлов, из которого макрос выбирает файлы для определения каждой переменной. Но если мы попробуем использовать этот макрос, мы получим ошибку:

```
$ make
Makefile:7: *** missing separator. Stop.
```

Этот способ не работает так, как ожидалось, из-за алгоритма работы синтаксического анализатора *make*. Макрос, вычисляемый на верхнем уровне, результат подстановки которого занимает больше одной строки, считается некорректным и приводит к синтаксической ошибке. В нашем случае анализатор ожидает, что строка является правилом или частью сценария сборки, но не обнаруживает подходящего разделителя. Довольно непонятное сообщение об ошибке. Функция *eval* была создана для решения этой проблемы. Если мы заменим нашу строку, содержащую *call*, на следующую:

```
$(eval $(call program-variables, ls, ls.c ls.h glob.c glob.h))
```

то получим именно то, что ожидали:

```
$ make
# ls.c glob.c
# ls.h glob.h
# ls.o glob.o
```

Использование *eval* решает проблему с синтаксическим анализатором, так как функция *eval* обрабатывает многострочные значения макросов и возвращает пустую строку.

Теперь у нас есть макрос, который определяет три переменных и не требует набора большого количества текста вручную. Заметим, что имена определяемых переменных в макросе составляются с помощью префикса, переданного в качестве аргумента, и фиксированного суффикса, `$1_sources`. Это напоминает технику переменных с вычисляемыми именами, описанную ранее.

Продолжая этот пример, можно заметить, что правила также могут быть заключены в макрос:

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $1_sources = $(filter %.c,$2)
  $1_headers = $(filter %.h,$2)
  $1_objects = $(subst .c,.o,$(filter %.c,$2))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
```

Обратите внимание, как две версии функции *program-variables* иллюстрируют проблему пробелов в аргументах функции. В предыдущей версии использование аргументов функции было устойчивым к начальным пробелам, то есть поведение

кода не изменялось при добавлении начальных пробелов к переменным `$1` и `$2`. Однако в новой версии были введены вычисляемые имена переменных `$($1_objects)` и `$($1_headers)`. Теперь добавление пробелов к первому аргументу функции (`ls`) приведёт к тому, что имена переменных будут вычислены неправильно, вследствие чего вместо списка файлов будет подставлена пустая строка. Обычно обнаружить подобную ошибку довольно сложно.

Когда мы запустим на выполнение наш *makefile*, обнаружится, что *make* игнорирует некоторые *.h* реквизиты. Чтобы понять причину этой проблемы, исследуем внутреннюю базу данных *make*, использовав ключ `--print-data-base`:

```
$ make --print-database | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c
ls: ls.o
```

Все *.h* реквизиты для *ls.o* пропущены! В правило, использующее вычисляемые имена переменных, вкралась какая-то ошибка.

Когда *make* производит синтаксический анализ вызова функции *call*, сначала производится вычисление функции, определённой пользователем, *program-variables*. После подстановки первая строка макроса выглядит так:

```
ls_sources = ls.c glob.c
```

Заметим, что каждая строка макроса вычисляется сразу же, как и ожидалось. Остальные определения переменных обрабатываются также. Затем происходит обработка правила:

```
$($1_objects): $($1_headers)
```

Сначала вычисляются имена переменных:

```
$(ls_objects): $(ls_headers)
```

Затем происходит подстановка значений переменных, что порождает следующее выражение:

```
:
```

Подождите! Куда делись значения наших переменных? Дело в том, что предыдущие три определения были правильно вычислены, но не были интерпретированы *make*. Давайте посмотрим, как это работает. Как только был произведён вызов *program-variables*, *make* «видит» следующее:

```
$(eval ls_sources = ls.c glob.c
ls_headers = ls.h glob.h
ls_objects = ls.o glob.o

:)
```

Затем выполняется функция *eval*, после чего определяются три переменные. Итак, причина найдена: переменные в правиле используются до того, как были определены.

Мы можем решить эту проблему, указав явно на необходимость отложить вычисление переменных, используемых в правиле, до момента их определения. Это можно сделать, экранировав знак доллара в вызове переменных с вычисляемым именем:

```
$$($1_objects): $$($1_headers)
```

На этот раз база данных *make* содержит те реквизиты, которые мы ожидали:

```
$ make -p | grep ^ls
ls_headers = ls.h glob.h
ls_sources = ls.c glob.c
ls_objects = ls.o glob.o
ls.c:
ls.o: ls.c ls.h glob.h
ls: ls.o
```

Итак, аргумент функции *eval* вычисляется дважды: сначала при подготовке списка аргументов для *eval*, затем ещё раз синтаксическим анализатором.

Мы решили предыдущую проблему, отложив вычисление переменных. Вторым способом решения этой проблемы является обёртка каждого определения переменной дополнительным вызовом функции *eval*:

```
# $(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))

  $($1_objects): $($1_headers)
endef

ls: $(ls_objects)
    $(eval $(call program-variables,ls,ls.c ls.h\
glob.c glob.h))
```

Заключая каждое определене переменной в собственный вызов *eval*, мы форсируем их вычисление в процессе подстановки макроса *program-variables*, поэтому их можно использовать непосредственно в теле макроса.

Лишь только мы улучшили наш *makefile*, нашлось ещё одно правило, которое можно добавить в макрос. Исполняемый файл программы зависит от объектных файлов. Поэтому, в завершение нашего параметризованного *makefile*'а, мы добавим цель `all` и переменную, содержащую список программ, сборкой которых может управлять наш *makefile*:

```
#$(call program-variables,variable-prefix,file-list)
define program-variables
  $(eval $1_sources = $(filter %.c,$2))
  $(eval $1_headers = $(filter %.h,$2))
  $(eval $1_objects = $(subst .c,.o,$(filter %.c,$2)))

  programs += $1

  $1: $($1_objects)

  ($1_objects): $($1_headers)
endef

# Объявляем цель по умолчанию
all:

$(eval $(call program-variables,ls,ls.c ls.h glob.c glob.h))
$(eval $(call program-variables,cp,...))
$(eval $(call program-variables,mv,...))
$(eval $(call program-variables,ln,...))
$(eval $(call program-variables,rm,...))

# Определяем реквизиты цели по умолчанию
all: $(programs)
```

Обратите внимание на расположение цели `all` и её реквизитов. Переменная `programs` не имеет правильного значения до осуществления пяти вызовов *eval*, но мы хотели бы объявить `all` целью по умолчанию, поместив её объявление первым в *makefile*'е. Мы можем удовлетворить всем ограничениям, объявив цель `all` первой и добавив реквизиты позже.

Функция *program-variables* имеет недостатки, поскольку некоторые переменные вычисляются слишком рано. Для решения этой проблемы *make* предоставляет функцию *value*, которая получает имя переменной в качестве аргумента и возвращает её значение, не осуществляя подстановок. Результат выполнения *value* может быть передан для обработки функции *eval*. Возвращая значение переменной, в котором не осуществлены подстановки, мы можем избежать проблем, из-за которых нам пришлось экранировать знаки доллара в нашем макросе.

К несчастью, функцию *value* нельзя использовать в макросе *program-variables*, поскольку она следует принципу «всё или ничего», не осуществляя подстановки ни одной переменной в аргументе. Более того, *value* не принимает параметров (а

при обнаружении не производит с ними никаких действий), поэтому имя нашей программы и список файлов останутся нетронутыми.

Из-за этих ограничений вы не очень часто будете видеть функцию *value* на страницах этой книги.

4.3.2 Триггеры

Функции, определяемые пользователем — это просто переменные, содержащие текст. Функция *call* заменяет вхождения позиционных параметров `$1`, `$2` и т.д. значениями фактических параметров, то есть разрешает ссылки в значении переменной, если они существуют. Если переменная не содержит формальных параметров, *call* не обратит на это внимания. Более того, если переменная вообще не содержит текста, *call* не произведёт никаких действий. Не будет вывода предупреждений или сообщений об ошибках. Это может разочаровать, если вы случайно допустили опечатку в имени вызываемой функции, однако иногда это очень полезно.

По своей сути функции — воплощение идеи повторного использования кода. Чем чаще вы используете какую-то функцию, тем более тщательно стоит подойти к её реализации. Функции можно сделать более удобными для повторного использования, если добавить к ним триггеры. *Триггер* (*hook*) — это ссылка на функцию, которая может быть определена пользователем для осуществления его собственных задач в процессе выполнения общих операций.

Предположим, ваш *makefile* служит для сборки множества библиотек. На одних платформах вы предпочли бы запуск *ranlib*, на других больше подходит *chmod*. Вместо явного указания команд для этих операций вы можете написать функцию и добавить к ней триггер:

```
# $(call build-library, object-files)
define build-library
  $(AR) $(ARFLAGS) $@ $1
  $(call build-library-hook,$@)
endef
```

Для использования триггера объявите функцию *build-library-hook*:

```
$(foo_lib): build-library-hook = $(RANLIB) $1
$(foo_lib): $(foo_objects)
    $(call build-library,$^)
$(bar_lib): build-library-hook = $(CHMOD) 444 $1
$(bar_lib): $(bar_objects)
    $(call build-library,$^)
```

4.3.3 Передача аргументов

Функция может получать информацию из четырёх «источников»: параметры, переданные с вызовом *call*, глобальные переменные, автоматические переменные

и переменные, зависящие от цели. Наиболее отвечающим принципу модульности способом передачи информации являются формальные параметры, поскольку они предохраняют функцию от изменений глобальных данных, однако иногда модульность — не самый важный критерий.

Предположим, у вас есть несколько проектов, использующих общий набор функций *make*. Каждый проект может идентифицироваться префиксом переменных, допустим, `PROJECT1_`, и имена всех критические важных переменных составляются из этого префикс и общих для всех проектов суффиксов. В предыдущих примерах вместо переменных наподобие `PROJECT_SRC` могли бы использоваться переменные `PROJECT1_SRC`, `PROJECT1_BIN` и `PROJECT1_LIB`. Вместо написания функции, которая требует наличия всех этих переменных мы могли бы использовать переменные с вычисляемым именем и передавать дополнительный аргумент — префикс:

```
# $(call process-xml,project-prefix,file-name)
define process-xml
  $($1_LIB)/xmlto -o $($1_BIN)/xml/$2 $($1_SRC)/xml/$2
endef
```

Другим подходом передачи аргументов является использование переменных, зависящих от цели. Это особенно удобно, когда бóльшая часть вызовов использует стандартные значения, и лишь в некоторых случаях требуется уделить этому особое внимание. Переменные, зависящие от цели, предоставляют также гибкость в том случае, когда правило определено в подключаемом файле, а выполнение правила осуществляется в *makefile*'е, в котором определены переменные.

```
release: MAKING_RELEASE = 1
release: libraries executables
...
$(foo_lib):
$(call build-library,$^)
...
# $(call build-library, file-list)
define build-library
  $(AR) $(ARFLAGS) $@           \
    $(if $(MAKING_RELEASE),     \
      $(filter-out debug/%,$1), \
  $1)
endef
```

Предыдущий код использует переменные, зависящие от цели, чтобы сообщить вызываемым функциям, что должна быть выполнена чистовая сборка библиотеки. В этом случае функция сборки не будет включать в библиотеку отладочные модули.

Глава 5

Команды

Мы уже рассмотрели бóльшую часть базовых концепций команд *make*, однако давайте всё же проведём их краткий обзор.

Команды являются однострочными сценариями командного интерпретатора. *make* считывает каждую команду и передаёт её в дочерний процесс командного интерпретатора для выполнения. На самом деле *make* может осуществлять оптимизацию этого (относительно) ресурсоёмкого fork/exec алгоритма, если это гарантированно не изменит поведения программы. Для этого каждая команда сканируется на наличие специальных символов командного интерпретатора, таких как шаблоны или перенаправления ввода-вывода. Если эти символы не найдены, *make* выполняет команду самостоятельно без порождения дочернего процесса.

В качестве командного интерпретатора по умолчанию используется */bin/sh*. За используемый интерпретатор отвечает переменная `SHELL`, не наследуемая из окружения. При старте *make* импортирует все переменные, кроме `SHELL`, из окружения пользователя, преобразуя их в переменные *make*. Это сделано для того, чтобы пользовательский выбор интерпретатора не вызвал ошибки выполнения *makefile*'а (возможно, входящего в загруженный из сети пакет программного обеспечения). Если пользователь действительно хочет поменять интерпретатор по умолчанию, он должен явно присвоить переменной `SHELL` нужное значение прямо в *makefile*'е. Мы вернёмся к этой теме в разделе «Выбор командного интерпретатора» этой главы.

5.1 Синтаксический анализ команд

Строки, начинающиеся с символа табуляции и следующие за спецификацией цели, считаются командами (если только предыдущая строка не завершалась символом обратного слэша). Встретив символ табуляции в другом контексте, GNU *make* старается догадаться, что вы имели ввиду. Например, присваивания перемен-

ных, комментарии и директивы включения могут начинаться с символа табуляции, если это не приводит к неоднозначности. Если *make* обнаруживает команду, которая не следует за спецификацией цели, выводится сообщение об ошибке:

```
makefile:20: *** commands commence before first target. Stop.
```

Формулировка этого сообщения немного удивляет, ведь оно часто появляется в середине *makefile*'а, много позже спецификации «первой» цели, однако теперь мы в состоянии понять это сообщение без особых проблем. Пожалуй, лучшей формулировкой была бы следующая: «обнаружена команда вне контекста определения правила».

Когда синтаксический анализатор обнаруживает команду в подходящем контексте, он включает «режим разбора команды», строя сценарий сборки по строке за раз. Добавление текста к сценарию прекращается, когда обнаруживается строка, не могущая быть частью сценария. В сценарии могут появляться следующие строки:

- Строки, начинающиеся с символа табуляции, являются командами, предназначенными для выполнения в командном интерпретаторе. Даже строки, которые могли бы быть интерпретированы как конструкции *make* (например, директивы `ifdef`, `include` или комментарии), в режиме разбора команд трактуются как команды.
- Пустые строки игнорируются, их выполнение не осуществляется.
- Строки, начинающиеся с символа `#`, возможно, с начальными пробелами (не символами табуляции!), воспринимаются как комментарии *make* и игнорируются.
- Директивы условной обработки, такие как `ifdef` и `ifeq`, распознаются и правильно обрабатываются в контексте сценария сборки.

Встроенные функции *make*, не предваряемые символом табуляции, не приводят к выходу из режима разбора команд. Это значит, что их значением должны быть допустимые команды интерпретатора или пустая строка. Значением функций *warning* или *eval* является пустая строка.

Тот факт, что пустые строки или комментарии *make* допустимы в сценариях сборки, поначалу может удивлять. Следующий пример показывает, как это работает:

```
long-command:
    @echo Строка 2:  далее пустая строка

    @echo Строка 4:  далее комментарий shell
```

```
    # Комментарий командного интерпретатора \
      (начинается с символа табуляции)
    @echo Строка 6:  далее комментарий make
# комментарий make в начале строки
    @echo Строка 8:  далее комментарий make
  # выровненный пробелами комментарий make
    # ещё один выровненный пробелами комментарий make
    @echo Строка 11: далее директива make
  ifdef COMSPEC
    @echo мы работаем под Windows
  endif
    @echo Строка 15: далее <<команда>> warning
    $(warning предупреждение)
    @echo Строка 17: далее <<команда>> eval
    $(eval $(shell echo Вывод shell 1>&2))
```

Заметим, что строки 5 и 10 выглядят очень похоже, но по сути они очень разные. Строка 5 содержит комментарий командного интерпретатора, начинающийся с символа табуляции, в то время как строка 10 содержит комментарий *make*, начинающийся с отступа в 4 пробела. Очевидно, что форматирование своих *makefile*'ов подобным образом — не самая лучшая идея (если только вы не собираетесь участвовать в конкурсе самых запутанных *makefile*'ов). Как вы можете видеть из следующего листинга, комментарии *make* не выполняются и не выводятся, даже если они появляются в контексте сценария сборки:

```
$ make
makefile:2: предупреждение
Вывод shell
Строка 2:  далее пустая строка
Строка 4:  далее коментарий shell
# комментарий командного интерпретатора (начинается...)
Строка 6:  далее комментарий make
Строка 8:  далее комментарий make
Строка 11: далее директива make
мы работаем под Windows
Строка 15: далее <<команда>> warning
Строка 17: далее <<команда>> eval
```

Сначала может показаться, что вывод функций *warning* и *eval* появился слишком рано, однако это не так (мы обсудим порядок вычисления в этой главе в разделе «Выполнение команд)». Тот факт, что сценарий сборки может содержать произвольное число пустых строк и комментариев, может быть источником трудно находимых ошибок. Предположим, вы случайно добавили строку с начальным символом табуляции. Если выше неё располагается определение цели, за которым следуют только комментарии и пустые строки, *make* будет трактовать вашу строку со случайным символом табуляции как команду, ассоциированную с предыдущей

целью. Как вы уже видели, это вполне допустимо и не вызовет ошибки или предупреждения, пока та же цель не встретится в составе другого правила в другом месте *makefile*'а (или во включаемом файле).

Если вам повезёт, ваш *makefile* будет содержать непустую строку, не содержащую комментария, между вашей ошибочной строкой и предыдущим сценарием сборки. В этом случае вы получите сообщение «commands commence before first target».

Сейчас самое время упомянуть об инструментальных средствах. Я думаю, теперь все согласны с тем, что использование символа табуляции для обозначения команды было не самым удачным решением, однако теперь уже поздно что-либо менять. Использование современного, понимающего синтаксис текстового редактора может помочь вам предотвратить потенциальные проблемы с помощью цветового выделения сомнительных конструкций. GNU *emacs* имеет довольно удобный режим для редактирования *makefile*'ов. Этот режим осуществляет подсветку синтаксиса и находит простые синтаксические ошибки, такие как пробелы после символа переноса строки или смешанные символы табуляции и пробелы. Мы вернёмся к теме совместного использования редактора *emacs* и *make* немного позже.

5.1.1 Продолжение длинных команд

Поскольку каждая команда выполняется в отдельном процессе командного интерпретатора, последовательности выражений, которые должны выполняться совместно, должны обрабатываться особым образом. Предположим для примера, что нам нужно создать файл, содержащий список файлов. Компилятор Java принимает такие файлы в случае, если нужно скомпилировать много исходного кода. Для этой цели мы можем написать следующее правило:

```
.INTERMEDIATE: file_list

file_list:
    for d in logic ui
    do
      echo $d/*.java
    done > $@
```

Очевидно, что этот пример не будет работать и вызовет ошибку при запуске:

```
$ make
for d in logic ui
/bin/sh: -c: line 2: syntax error: unexpected end of file
make: *** [file_list] Error 2
```

В качестве решения можно попробовать добавить символы продолжения в конце каждой строки:

```
.INTERMEDIATE: file_list

file_list:
    for d in logic ui \
    do                \
      echo $d/*.java  \
    done > $@
```

Однако теперь при запуске появляется другая ошибка:

```
$ make
for d in logic ui \
do                \
  echo /*.java    \
done > file_list
/bin/sh: -c: line 1: syntax error near unexpected token `>'
/bin/sh: -c: line 1: `for d in logic ui do       echo /*.java
make: *** [file_list] Error 2
```

Что же произошло? В коде есть две ошибки. Во-первых, ссылка на переменную цикла, `d`, должна быть экранирована. Во-вторых, поскольку цикл передаётся в интерпретатор одной строкой, мы должны добавить точку с запятой после списка файлов и выражения в теле цикла:

```
.INTERMEDIATE: file_list

file_list:
    for d in logic ui; \
    do                 \
      echo $d/*.java;  \
    done > $@
```

Теперь мы получим именно то, что ожидали. Поскольку цель `file_list` помечена как `.INTERMEDIATE`, *make* удалит её после завершения компиляции.

В более реалистичном примере список файлов будет храниться в переменной *make*. Если есть уверенность в том, что этот список достаточно мал, мы можем осуществить ту же операцию без использования цикла, используя только встроенные функции *make*:

```
.INTERMEDIATE: file_list

file_list:
    echo $(addsuffix /*.java,$(COMPILATION_DIRS)) > $@
```

Однако у версии с циклом меньше шансов столкнуться с проблемой конечности длины командной строки, если список каталогов будет расти со временем.

Ещё одной общей проблемой является смена текущего каталога:

```
TAGS:
    cd src
    ctags --recurse
```

Очевидно, что предыдущий пример не выполнит программу *ctags* в каталоге *src*. Чтобы добиться желаемого эффекта, мы должны либо разместить оба выражения в одной строке, либо экранировать символ новой строки (разделив выражения точкой с запятой):

```
TAGS:
    cd src; \
    ctags --recurse
```

Ещё более разумно проверять статус выполнения программы *cd* перед выполнением *ctags*:

```
TAGS:
    cd src && \
    ctags --recurse
```

Заметим, что при некоторых обстоятельствах можно опустить точку с запятой, не вызвав ошибки командного интерпретатора или *make*:

```
disk-free = echo "Проверяем размер дискового пространства..." \
    df . | awk '{ print $$4 }'
```

Этот пример выводит простое сообщение, за которым следует число свободных блоков на текущем устройстве. Или нет? Мы случайно забыли поставить точку с запятой после команды *echo*, в результате чего программа *df* никогда не будет запущена. Вместо этого мы направим сообщение «`Проверяем размер дискового пространства... df .`» на вход программы *awk*, которая напечатает четвёртое поле строки, т.е. `пространства...`.

Возможно, вам приходилось использовать директиву `define`, предназначенную для создания многострочных последовательностей. К сожалению, она не решает проблемы переносов строк. Когда происходит подстановка макроса, каждая его строка вставляется в сценарий сборки с начальным символом табуляции, и *make* работает с каждой строкой независимо. Разные строки макроса выполняются в разных экземплярах командного интерпретатора, поэтому вам следует обращать внимание на перенос строк даже в определениях макросов.

5.1.2 Модификаторы команд

Команды могут быть модифицированы при помощи нескольких префиксов. Мы уже встречались с «молчаливым» префиксом, `@`, ниже приведён полный список возможных префиксов с некоторыми комментариями:

`@`

Подавляет вывод команды. Для исторической совместимости вы можете поместить вашу команду в реквизиты специальной цели `.SILENT`, если хотите, чтобы все команды, ассоциированные с вашей целью, были скрыты. Однако использование `@` предпочтительней, поскольку этот модификатор может быть применён к отдельным командам в сценарии. Если вы хотите применить модификатор ко всем целям, используйте опцию `--silent` (или просто `-s`).

Сокрытие команд может сделать вывод *make* приятнее для глаза, однако это также ведёт к затруднениям при отладке. Если вы обнаружите, что часто удаляете модификатор `@`, а затем возвращаете его на место, разумно завести переменную (к примеру, `QUIET`), содержащую модификатор, и использовать её в командах:

```
QUIET = @
hairy_script:
    $(QUIET) сложный сценарий ...
```

В этом случае при необходимости увидеть сложный сценарий в том виде, в котором его выполняет *make*, просто сбросьте значение переменной `QUIET` в командной строке:

```
$ make QUIET= hairy_script
сложный сценарий ...
```

`-`

Этот префикс сообщает *make*, что ошибки, возникающие при выполнении помеченной команды, следует игнорировать. По умолчанию *make* проверяет код возврата каждой программы или конвейера программ. Если какая-то программа завершается с ненулевым кодом, *make* прерывает выполнение оставшейся части сценария и завершает своё выполнение. Этот модификатор заставляет *make* игнорировать код возврата модифицированной строки и продолжать выполнение в обычном режиме. Мы обсудим эту тему подробнее в следующем разделе.

Для исторической совместимости вы можете игнорировать ошибки в части сценария сборки, поместив соответствующую цель в реквизиты специальной цели `.IGNORE`. Вы можете игнорировать все ошибки в *makefile*'е используя опцию `--ignore-errors` (или `-i`). Однако полезность этой возможности вызывает сомнения.

`+`

Этот модификатор сообщает *make*, что помеченная им команда должна выполняться даже в том случае, если в опциях командной строки встречается `--just-print` (или `-n`). Этот модификатор используется при составлении рекурсивных *makefile*'ов. Мы обсудим эту тему в разделе «Рекурсивный *make*» главы 6.

Все модификаторы должны встречаться в строке только один раз. Очевидно, перед выполнением команд модификаторы вырезаются.

5.1.3 Ошибки и прерывания

Каждая команда, выполняемая *make*, возвращает код ошибки. Нулевой код соответствует успешному выполнению команды, ненулевой — ошибочной ситуации. Некоторые программы используют код возврата для передачи более полезной информации, чем просто факта возникновения ошибки. Например, программа *grep* возвращает 0 в случае обнаружения соответствия шаблону, 1 в случае отсутствия соответствий и 2 в случае возникновения ошибки какого-либо рода.

Обычно при ошибке выполнения команды (т.е. при возвращении ненулевого кода ошибки) *make* прекращает выполнение команд и завершает выполнение с ненулевым кодом возврата. Иногда нужно, чтобы *make* продолжал работу, собрав столько целей, сколько возможно. Например, вам может понадобиться скомпилировать максимально возможное число исходных файлов, чтобы увидеть все ошибки компиляции в один проход. Вы можете добиться такого поведения при помощи опции `--keep-going` (или `-k`).

Поскольку модификатор `-` заставляет *make* игнорировать ошибки в отдельных командах, я стараюсь избегать его использования, поскольку это усложняет автоматическую обработку ошибок и привносит в код небрежность.

Когда *make* игнорирует ошибку, выводится сообщение об ошибке с именем цели в квадратных скобках. Ниже приведён вывод, полученный при попытке удаления несуществующего файла:

```
rm non-existent-file
rm: cannot remove ‘non-existent-file’: No such file or directory
make: [clean] Error 1 (ignored)
```

Некоторые команды (например, *rm*) имеют опции для подавления ошибочных кодов возврата. Опция `-f` заставит программу *rm* вернуть нулевой код ошибки и подавить вывод предупреждений. Использование этой опции лучше, чем зависимость от модификатора.

Бывают случаи, когда возврат программой нулевого кода ошибки считается неудачей и наоборот. В таких случаях можно просто инвертировать код возврата программы:

```
# Убедимся, что в коде не осталось отладочных сообщений.
.PHONY: no_debug_printf

no_debug_printf: $(sources)
    ! grep --line-number '"debug:' $^
```

К сожалению, версия GNU *make* 3.80 содержит ошибку, мешающую непосредственному использованию этой возможности: *make* не распознаёт символ `!` как часть команды, требующей вызова командного интерпретатора, и выполняет оставшуюся часть строки самостоятельно, вызывая ошибку. В качестве простого обхода проблемы можно использовать в команде специальные символы как намёк для *make*:

```
# Убедимся, что в коде не осталось отладочных сообщений.
.PHONY: no_debug_printf

no_debug_printf: $(sources)
    ! grep --line-number '"debug:' $^ < /dev/null
```

Другим источником неожиданных ошибок в командах является оператор `if` без ветки `else`:

```
$(config): $(config_template)
    if [ ! -d $(dir $@) ];  \
    then                    \
        $(MKDIR) $(dir $@); \
    fi
    $(M4) $^ > $@
```

Первая команда проверяет, существует ли целевой каталог и в случае необходимости вызывает программу *mkdir* для его создания. К сожалению, если каталог существует, команда `if` вернёт ненулевой код ошибки (код возврата программы *test*), что приведёт к завершению работы сценария. Одним из решений этой проблемы является добавление ветви `else`:

```
$(config): $(config_template)
    if [ ! -d $(dir $@) ];  \
    then                    \
        $(MKDIR) $(dir $@); \
    else                    \
        true;               \
    fi
    $(M4) $^ > $@
```

Двоеточие (`:`) — это команда интерпретатора, которая всегда возвращает истину, поэтому её можно использовать вместо `true`. Альтернативной рабочей реализацией является следующая:

```
$(config): $(config_template)
    [[ -d $(dir $@) ]] || $(MKDIR) $(dir $@)
    $(M4) $^ > $@
```

Теперь первое выражение истинно, если целевой каталог существует или выполнение программы *mkdir* завершилось успешно. Другой альтернативой является использование ключа `-t` программы *mkdir*. Это приведёт к успешному завершению *mkdir* даже в том случае, если требуемый каталог уже сужествует.

Все предыдущие реализации вызывали интерпретатор даже в том случае, если каталог уже существовал. Использование функции *wildcard* позволяет избежать выполнения команд в случае наличия каталога:

```
# $(call make-dir, directory)
make-dir = $(if $(wildcard $1),,$(MKDIR) -p $1)

$(config): $(config_template)
    $(call make-dir, $(dir $@))
    $(M4) $^ > $@
```

Поскольку каждая команда выполняется в отдельном экземпляре командного интерпретатора, общей практикой является использование многострочных команд, разделённых точками с запятой. Остерегайтесь случаев, когда ошибка в таких сценариях может не привести к завершению выполнения сборки:

```
target:
    rm rm-неудачен; echo но следующая команда выполняется
```

Лучшей практикой является минимизация длины сценария, что даёт *make* шанс обработать код возврата самостоятельно. Например:

```
path-fixup = \
    -e "s;[a-zA-Z:/]*/src/;$(SOURCE_DIR)/;g" \
    -e "s;[a-zA-Z:/]*/bin/;$(OUTPUT_DIR)/;g"

# хорошая версия
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed && \
  mv $2.fixed $2
endef

# отличная версия
define fix-project-paths
  sed $(path-fixup) $1 > $2.fixed
  mv $2.fixed $2
endef
```

Этот макрос преобразует пути в стиле DOS (с прямыми слэшами) в пути назначения для определённой структуры каталогов исходного кода и бинарных файлов. Макрос принимает имена двух файлов: входного и выходного. Кроме того, принимаются дополнительные действия, чтобы выходной файл был перезаписан только в том случае, если *sed* завершится корректно. В «хорошей» версии это достигается за счёт соединения программ *sed* и *mv* операцией `&&`, в результате чего они выполняются в одном экземпляре командного интерпретатора. «Лучшая» версия выполняет их как две отдельных команды, позволяя *make* завершить выполнение сценария, если программа *sed* завершится неудачей. Однако «лучшая» версия не является менее производительной (программа *mv* не требует вызова командного интерпретатора и выполняется непосредственно *make*), к тому же её легче понять, а в случае возникновения ошибки полученное сообщение будет более информативным (поскольку *make* укажет, какая именно команда закончилась неудачей).

Заметим, что предыдущая ситуация не имеет отношения к общей проблеме команды *cd*:

```
TAGS:
    cd src && \
    ctags --recurse
```

В этом случае оба выражения должны выполняться в одном процессе командного интерпретатора, поэтому должен использоваться разделитель команд, например, `;` или `&&`.

Удаление и сохранение целевых файлов

Если происходит ошибка, *make* подразумевает, что повторная сборка цели не может быть осуществлена. В этом случае любая другая цель, имеющая текущую в качестве реквизита, также не может быть собрана, поэтому *make* не будет даже пытаться осуществить её сборку. Если использована опция `--keep-going` (`-k`), будет произведена попытка сборки следующей цели, иначе *make* закончит своё выполнение. Если текущей целью является файл, его содержимое может быть повреждено, если команда из сценария завершится, не закончив своей работы. К сожалению, *make* оставит этот потенциально повреждённый файл на диске из соображений исторической совместимости. Поскольку время последнего изменения файла будет изменено, все последующие вызовы *make* не смогут поместить в этот файл корректные данные. Вы можете избежать этой проблемы и заставить *make* удалять эти подозрительные файлы в случае возникновения ошибки, указав целевой файл реквизитом специальной цели `.DELETE_ON_ERROR`. Если цель `.DELETE_ON_ERROR` используется без реквизитов, ошибка при сборке любого файла приведёт к его удалению.

Дополнительные проблемы связаны с ситуацией, когда выполнение *make* прерывается сигналом, например, по нажатию клавиш Ctrl-C. В этом случае *make*

удалит текущий целевой файл, если он был модифицирован. Иногда удаление целевого файла не является желаемой реакцией. Возможно, создание целевого файла — чрезвычайно затратная операция, или получение части его содержимого желательней полного его отсутствия. В таком случае вы можете защитить целевой файл, сделав его реквизитом специальной цели `.PRECIOUS`.

5.2 Выбор командного интерпретатора

Когда *make* требуется передать команду интерпретатору, он использует */bin/sh*. Вы можете изменить интерпретатор, выставив соответствующим образом значение переменной `SHELL`. Однако хорошенько подумайте перед тем, как это сделать. Обычно назначением *make* является предоставление для команды разработчиков инструмента сборки системы из исходного кода. Довольно легко создать *makefile*, не соответствующий этому назначению, используя инструменты, не доступные для других участников процесса разработки или строя предположения, для них не справедливые. Использование любых интерпретаторов, отличных от */bin/sh*, считается дурным тоном для любого широко распространённого приложения (доступного через ftp или открытый репозиторий cvs). Мы обсудим вопросы переносимости более детально в главе 7.

Однако и есть другой контекст использования *make*. В закрытых средах разработки часто все участники проекта работают на ограниченном множестве машин и операционных систем. На самом деле это именно та среда, в которой мне чаще всего приходилось работать. В этой ситуации вы имеете полное право настроить среду, в которой будет работать *make*, по своему усмотрению. Следует лишь инструктировать всех разработчиков в вопросах настройки среды и работы со сборками.

В подобных средах я предпочитаю открыто жертвовать переносимостью в некоторых аспектах. Я уверен, что это может сделать процесс разработки гораздо более гладким. Одной их таких жертв является замена стандартного значения переменной `SHELL` на */usr/bin/bash*. *bash* — это переносимый, POSIX-совместимый командный интерпретатор (отсюда следует, что он включает в себя все возможности *sh*), являющийся интерпретатором по умолчанию для GNU/Linux. Причиной многих проблем с переносимостью *makefile*'ов является использование непереносимых конструкций в сценариях сборки. Решением этих проблем является явное использование одного стандартного интерпретатора вместо употребления лишь переносимого подмножества команд *sh*. У Пола Смита, разработчика, занимающегося поддержкой GNU *make*, ести веб-страница «Правила Пола для *makefile*'ов» («Paul's Rules of Makefiles», *http://make.paulandlesley.org/rules.html*), на которой он делает следующее замечание: «Не тратьте силы, пытаясь написать переносимые *makefile*'ы, используйте переносимую версию *make*!» («Don't hassle with

writing portable makefiles, use portable make instead!»). Я могу добавить следующее: «Когда есть возможность, не тратьте силы, пытаясь написать переносимый сценарий, используйте переносимый командный интерпретатор (bash)». *bash* работает на большинстве операционных систем, включая практически все варианты Unix, Windows, BeOS, Amiga и OS/2.

В оставшейся части книги я буду явно указывать на случаи использования специфичных возможностей *bash*.

5.3 Пустые команды

Пустая команда – это команда, которая не производит никаких действий:

```
header.h: ;
```

Вспомним, что за списком реквизитов цели может следовать точка с запятой и команда. Здесь используется только точка с запятой, что означает, что команды не предполагаются. Вместо этого вы можете поместить после определения цели строку, содержащую только один символ табуляции, однако это будет невозможно прочитать. Пустые команды чаще всего используются для предотвращения соответствия цели шаблонному правилу и выполнения нежелательных команд.

Заметим, что в других версиях *make* пустые цели иногда используются в качестве абстрактных. В GNU *make* следует использовать специальную цель `.PHONY`, это безопасней и яснее.

5.4 Команды и окружение

Команды, выполняемые *make*, наследуют окружение процесса *make*. Это окружение включает текущий каталог, файловые дескрипторы и переменные окружения, передаваемые *make*.

Когда создаётся дочерний процесс командного интерпретатора, *make* добавляется в окружение несколько переменных:

```
MAKEFLAGS
MFLAGS
MAKELEVEL
```

Переменная `MAKEFLAGS` включает опции командной строки, переданные *make*. Переменная `MFLAGS` дублирует содержимое `MAKEFLAGS` и существует по историческим причинам. Переменная `MAKELEVEL` содержит число вложенных вызовов *make*. Таким образом, когда *make* рекурсивно вызывает *make*, переменная `MAKELEVEL` увеличивается на единицу. Подпроцесс родительского процесса *make* будет иметь переменную `MAKELEVEL`, значением которой будет единица. Все эти переменные

обычно используются для управления рекурсивным *make*. Мы обсудим эту тему в разделе «Рекурсивный *make*» главы 6.

Конечно, пользователь может передать в окружение дочернего процесса любую переменную по своему усмотрению, используя директиву `export`.

Текущий рабочий каталог исполняемой команды совпадает с рабочим каталогом родительского процесса *make*. Обычно это тот же каталог, из которого была вызвана программа *make*, однако его можно заменить при помощи опции `-directory=`*`каталог`* (или `-C`). Заметим, что спецификация *makefile*'а при помощи опции `--file` не изменяет рабочий каталог, только устанавливает *makefile*, который нужно прочитать.

Каждый подпроцесс, порождаемый *make*, наследует три стандартных файловых дескриптора: *stdin*, *stdout* и *stderr*. Здесь нет ничего особенного, за исключением одного следствия: сценарий сборки может считывать данные из стандартного потока ввода. Как только сценарий считает все данные из потока, оставшиеся команды выполняются в обычном порядке. Однако ожидается, что *makefile*'ы должны работать корректно без этого типа взаимодействия. Пользователь часто расчитывает на возможность просто запустить *make* и далее не принимать никакого участия в процессе сборки, проверив лишь результаты по завершению. И, конечно, сложно придумать полезное применение чтению стандартного потока ввода в контексте автоматизированных сборок, основанных на использовании *cron*.

Общей ошибкой является случайное чтение стандартного потока ввода:

```
$(DATA_FILE): $(RAW_DATA)
    grep pattern $(RAW_DATA_FILES) > $@
```

Здесь входные файлы для *grep* хранятся в переменной (при использовании которой произошла опечатка). Если вместо значения переменной подставится пустая строка, *grep* останется только читать данные со стандартного потока ввода, без каких либо объяснений причины «зависания» *make*. Простым способом избежать такой проблемы является включение в команду дополнительного файла устройства */dev/null*:

```
$(DATA_FILE): $(RAW_DATA)
    grep pattern $(RAW_DATA_FILES) /dev/null > $@
```

Такая команда никогда не примет попытки чтения стандартного потока ввода. Естественно, отладка *makefile*'ов также помогает избежать неприятностей.

5.5 Выполнение команд

Обработка командного сценария происходит в четыре этапа: чтение кода, подстановка переменных, вычисление выражений *make* и выполнение команд. Давайте посмотрим, как все эти этапы применяются к сложному сценарию. Рассмотрим следующий (немного надуманный) *makefile*. Приложение компонуется, затем

от полученного исполняемого файла отделяется таблица символов, после чего он сжимается при помощи компрессора исполняемых файлов *upx*:

```
# $(call strip-program, file)
define strip-program
  strip $1
endef

complex_script:
    $(CC) $^ -o $@
  ifdef STRIP
    $(call strip-program, $@)
  endif
    $(if $(PACK), upx --best $@)
    $(warning Final size: $(shell ls -s $@))
```

Вычисление командных сценариев откладывается до того момента, когда оно действительно потребуется, однако директивы `ifdef` обрабатывается сразу после их обнаружения. Поэтому *make* считывает команды сценария, игнорируя их содержимое и сохраняя каждую строку, пока не обнаружит строку `ifdef STRIP`. *make* выполняет тест, и если переменная `STRIP` не определена, *make* считывает и отбрасывает весь текст сценария, пока не натолкнётся на закрывающую директиву `endif`. После этого *make* считывает и сохраняет оставшуюся часть сценария.

Когда приходит время выполнения сценария, *make* сначала сканирует команды на наличие конструкций *make*, требующих подстановки. После подстановки макросов каждая строка сценария начинается с символа табуляции. Вычисление макросов *перед* выполнением команд может привести к неожиданным результатам. Последняя строка в нашем сценарии некорректна. Функции *shell* и *warning* выполняются *до* компоновки приложения. Поэтому команда *ls* будет выполнена до того, как целевой файл будет собран. Это объясняет «неправильный» порядок выполнения, который мы наблюдали в разделе «Синтаксический анализ команд».

Также заметим, что строка `ifdef STRIP` выполняется во время чтения *makefile*'а, однако строка `$(if...)` вычисляется непосредственно перед выполнения сценария сборки цели `complex_script`. Использование функции *if* допускает написание более гибких сценариев, поскольку предоставляет больше возможностей для контроля определения переменных, однако такой подход не очень хорошо приспособлен для управления большими блоками текста.

Как показывает наш пример, всегда очень важно обращать внимание на то, какая программа вычисляет выражение (т.е. *make* или командный интерпретатор), и когда именно это вычисление происходит:

```
$(LINK.c) $(shell find                              \
            $(if $(ALL),$(wildcard core ext*),core) \
              -name '*.o')
```

Это запутанный командный сценарий компоновки множества объектных файлов. Порядок вычисления операций таков (в скобках указана программа, выполняющая соответствующую операцию):

1. Вычисление `$ALL` (*make*).
2. Вычисление *if* (*make*).
3. Вычисление *wildcard* в предположении, что `ALL` содержит непустое значение (*make*).
4. Вычисление *shell* (*make*).
5. Вычисление *find* (*sh*).
6. После завершения подстановок и вычисления конструкций *make*, происходит выполнение команды компоновки (*sh*).

5.6 Ограничения командной строки

Во время работы с крупными проектами вы можете столкнуться с ограничениями на длину команд, которые *make* пытается выполнить. Ограничения на длину командной строки варьируются в зависимости от операционной системы. Red Hat 9 GNU/Linux позволяет выполнять команды длиной не более 128 Кб, а Windows XP ограничивает длину 32 Кб. Сообщения об ошибке также варьируются. Если вы вызовите команду *ls* со слишком большим списком параметров, в Cygwin под Windows, то получите следующее сообщение:

```
C:\usr\cygwin\bin\bash: /usr/bin/ls: Invalid argument
```

На Red Hat 9 сообщение выглядит иначе:

```
/bin/ls: argument list too long
```

Даже 32 Кб выглядит как довольно большой объём данных для командной строки, однако когда ваш проект содержит 3000 файлов и 100 подкаталогов, и вы хотите манипулировать ими всеми, это ограничение может быть довольно существенным.

Существует два основных пути, которые ведут к неприятностям с ограничениями на длину строки: вычисление базовых значений при помощи инструментов командного интерпретатора или использование *make* для присваивания переменной значения очень большой длины. Предположим для примера, что мы хотим скомпилировать все исходные файлы одной командой:

```
ompile_all:
$(JAVAC) $(wildcard $(addsuffix /*.java,$(source_dirs)))
```

Переменная *make* `source_dirs` может содержать всего несколько сотен слов, однако после добавления шаблона исходных файлов Java и применения функции *wildcard* этот список может превысить предельную длину командной строки вашей системы. Кстати, *make* не имеет собственных ограничений на длину строки, позволяя вам хранить столько данных, сколько может вместить виртуальная память.

Когда сталкиваешься с подобной ситуацией, возникает ощущение, что играешь в старую игру «Приключение» (Adventure): «Вы находитесь в лабиринте из одинаковых извилистых коридоров». Например, вы можете попробовать использовать *xargs* для решения проблемы, так как *xargs* разделяет строки на части в соответствии с ограничениями текущей системы:

```
compile_all:
    echo $(wildcard
           $(addsuffix /*.java,$(source_dirs))) | \
    xargs $(JAVAC)
```

К сожалению, так мы просто переместили проблему ограничений из команды *javac* в команду *echo*. Мы также не можем использовать *echo* или *printf* для записи данных в файл (предполагается, что компилятор может читать список файлов из файла).

Нет, для решения этой проблемы нужно в первую очередь избежать создания одного большого списка файлов. Вместо этого мы можем просматривать по одному каталогу за раз, используя шаблоны командного интерпретатора:

```
compile_all:
    for d in $(source_dirs); \
    do                       \
        $(JAVAC) $$d/*.java; \
    done
```

Также можно использовать канал в *xargs*, чтобы достигнуть желаемого результата за меньшее количество вызовов компилятора:

```
compile_all:
    for d in $(source_dirs); \
    do                       \
        echo $$d/*.java;     \
    done |                   \
    xargs $(JAVAC)
```

К сожалению, ни один из этих сценариев не обрабатывает должным образом ошибки компиляции. Лучшим подходом является сохранение полного списка файлов и последующая передача его компилятору, если, конечно, компилятор поддерживает чтение аргументов из файла. Компилятор Java поддерживает эту возможность:

```
compile_all: $(FILE_LIST)
    $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
    for d in $(source_dirs); \
    do                       \
        echo $$d/*.java;     \
    done > $@
```

Обратите внимание на тонкую ошибку в цикле `for`. Если какой-либо из каталогов не содержит исходных файлов Java, строка **.java* будет включена в список файлов и компилятор Java выдаст сообщение об ошибке: «File not found» (файл не найден). Мы можем приказать *bash* подставлять пустые строки на место шаблонов, которым не соответствует ни один файл, использовав опцию `nullglob`:

```
compile_all: $(FILE_LIST)
    $(JAVA) @$<

.INTERMEDIATE: $(FILE_LIST)
$(FILE_LIST):
    shopt -s nullglob;       \
    for d in $(source_dirs); \
    do                       \
        echo $$d/*.java;     \
    done > $@
```

Многим проектам приходится создавать список файлов. Ниже представлен макрос, содержащий сценарий *bash*, создающий список файлов. Первым аргументом является корневой каталог, пути всех найденных файлов будут указываться относительно этого каталога. Вторым параметром является список каталогов, в которых нужно искать файлы, соответствующие шаблону. Третий и четвёртый аргументы опциональны и содержат расширения интересующих файлов.

```
# $(call collect-names,root-dir,dir-list,
#        suffix1-opt,suffix2-opt)
define collect-names
  echo Making $@ from directory list...
  cd $1;                                              \
  shopt -s nullglob;                                  \
  for f in $(foreach file,$2,'$(file)'); do           \
    files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) ); \
    if (( $${#files[@]} > 0 ));                       \
    then                                              \
      printf '"%s"\n' $${files[@]};                   \
    else :; fi;                                       \
  done
endef
```

Так выглядит шаблонное правило создания списка файлов изображений:

```
%.images:
    @$(call collect-names,$(SOURCE_DIR),$^,gif,jpeg) > $@
```

Вычисление макроса скрыто с помощью модификатора `@`, поскольку сценарий достаточно велик, а причину для копирования и вставки полученного кода найти трудно. Список каталогов указан в реквизитах. После смены текущего каталога сценарий включает опцию `nullglob`. Остаток макроса — цикл *for*, осуществляющий проход по всем каталогам, которые нужно обработать. Первое выражение поиска файлов — это список слов, переданный в качестве второго параметра (`$2`). Сценарий экранирует слова в списке файлов с помощью апострофа, так как они могут содержать символы, имеющие для командного интерпретатора специальный смысл. В частности, имена файлов в некоторых языках программирования (например, Java) могут содержать символы доллара:

```
for f in $(foreach file,$2,'$(file)'); do
```

Мы производим поиск файлов в каталоге, заполняя массив `files` результатами вычислений подстановок. Если полученный массив содержит элементы, мы используем *printf* для того, чтобы напечатать каждое слово на новой строке. Использование массива позволяет макросу правильно обрабатывать пути, содержащие пробелы. Возможность наличия пробелов в путях — это ещё одна причина, по которой аргумент *printf* окружён кавычками.

Список файлов создаётся при помощи следующей строки:

```
files=( $$f$(if $3,/*.{$3$(if $4,$(comma)$4)}) );
```

Переменная `$$f` — это каталог или файл, переданный макросу в составе аргумента. Следующее выражение — это функция *if*, проверяющая третий аргумент на непустоту. Это один из путей, который можно использовать для реализации необязательных аргументов. Если третий аргумент пуст, четвёртый также подразумевается пустым. В этом случае файл, переданный пользователем, должен быть включён в список как есть. Это позволяет макросу строить списки обычных файлов, для которых использование шаблонов не подходит. Если третий аргумент не пуст, функция *if* добавляет к корневому каталогу строку `/*{$3}`. Если передан четвёртый аргумент, после `$3` происходит вставка `$4`. Обратите внимание на то, как происходит вставка запятой в шаблон. Поместив символ запятой в переменную *make*, мы можем незаметно передать её в выражение, явное использование запятой было бы воспринято как отделение *then* части от *else* части функции *if*. Определение переменной `comma` очевидно:

```
comma = ,
```

Все рассмотренные циклы *for* зависели от пределов длины командной строки, поскольку использовали шаблонные выражения. Разница в том, что результат применения шаблона для поиска файлов в одном каталоге имеет гораздо меньше шансов превысить предел.

Что будет, если какая-то переменная *make* содержит длинный список файлов? Чтож, тогда мы столкнулись с настоящей неприятностью. Я нашёл лишь два пути передать длинную переменную *make* в интерпретатор. Первый подход — передавать содержимое переменной по частям, используя фильтры, основанные на применении функции *wordlist*:

```
compile_all:
    $(JAVAC) $(wordlist 1, 499, $(all-source-files))
    $(JAVAC) $(wordlist 500, 999, $(all-source-files))
    $(JAVAC) $(wordlist 1000, 1499, $(all-source-files))
```

Второй путь — использовать функцию *filter*, однако в этом случае результаты менее предсказуемы, поскольку число отбираемых фильтром файлов может зависеть от числа слов, соответствующему каждому из выбранных шаблонов. В следующем примере используются шаблоны, основанные на алфавитном порядке:

```
compile_all:
    $(JAVAC) $(filter a%, $(all-source-files))
    $(JAVAC) $(filter b%, $(all-source-files))
```

Ваши шаблоны могут использовать специальные свойства имён файлов.

Обратите внимание на то, как сложно автоматизировать этот процесс. Мы могли бы попробовать использовать алфавитный подход совместно с циклом *foreach*:

```
compile_all:
    $(foreach l,a b c d e ...,                \
      $(if $(filter $l%, $(all-source-files)), \
        $(JAVAC) $(filter $l%, $(all-source-files));))
```

Однако такой подход не работает. *make* превратит этот сценарий в одну строку текста, что только усугубит проблемы с длиной команд. Вместо этого можно использовать *eval*:

```
compile_all:
    $(foreach l,a b c d e ...,                \
      $(if $(filter $l%, $(all-source-files)), \
        $(eval                                \
          $(shell                             \
            $(JAVAC) $(filter $l%, $(all-source-files));))))
```

Этот вариант будет работать правильно, потому что функция *eval* выполняет команду *shell* незамедлительно, и результатом её вычисления является пустая строка. Таким образом, результатом вычисления функции *foreach* является также пустая строка. Проблема заключается в том, что проверка ошибок в этом контексте не происходит, поэтому ошибки компиляции не будут переданы *make* напрямую.

Подход, основанный на использовании *wordlist* значительно хуже. Из-за ограниченных возможностей *make* в области численных операций, применить эту технику в цикле не получится. В общем, сколь-нибудь удовлетворительных техник обращения с огромными списками файлов практически не существует.

Часть II

Специализированные вопросы

Во второй части мы получим проблемно-ориентированный взгляд на *make*. Далеко не всегда бывает очевидно, как применить *make* к проблемам реального мира, таким как сборки в нескольких каталогах, новые языки программирования, переносимость, проблемы производительности и отладка. В этой части обсуждается каждая из этих проблем, в добавок здесь вы найдёте главу, в которой рассматривается несколько сложных примеров.

Глава 6

Управление большими проектами

Какой проект можно назвать большим? Для наших целей мы назовём большим проект, требующий команды разработчиков, поддерживающий несколько архитектур, предполагающий несколько релизов и нуждающийся в поддержке. Конечно, проект не должен иметь все эти признаки, чтобы называться большим. Миллион строк кода на C++ в предварительном релизе, предназначенных для одной платформы — это тоже большой проект. Однако программное обеспечение редко остаётся в стадии предварительного релиза навечно. Если оно успешно, в конечном итоге кто-то попросит перенести его на другую платформу. Поэтому все крупные системы программного обеспечения на определённом этапе становятся похожими.

Большие проекты обычно упрощаются при помощи декомпозиции на отдельные компоненты, которые обычно собираются в самостоятельные программы или библиотеки (или и то, и другое). Эти компоненты часто хранятся в отдельных каталогах файловой системы и управляются при помощи собственных *makefile*'ов. Один из способов сборки всей системы компонентов подразумевает наличие главного *makefile*'а, вызывающего *makefile*'ы компонентов в нужном порядке. Этот подход называется *рекурсивный make* (*recursive make*), потому что главный *makefile* вызывает *make* рекурсивно для обработки *makefile*'а каждого компонента. Рекурсивный *make* — это общая техника для компонентных сборок. Альтернатива, предложенная Питером Миллером (Peter Miller) в 1988 году, лишена многих недостатков, присущих рекурсивному *make*, и основана на использовании единственного *makefile*'а, включающего информацию из каталогов компонентов[1].

Как только проект проходит этап сборки компонентов, обычно на его пути встают более серьёзные организационные проблемы управления сборками, включающие управление разработкой нескольких версий проекта, поддержку нескольких

[1]Miller, P.A., *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25(1998). Эта статья также доступна по адресу `http://aegis.sourceforge.net/auug97.pdf` (прим. автора).

```
.
|
|--makefile
|
|--include
|  |--db
|  |--codec
|  `--ui
|
|--lib
|  |--db
|  |--codec
|  `--ui
|
|--app
|  `--player
|
`--doc
```

Рис. 6.1: Структура каталогов проекта mp3 плеера

платформ, предоставление эффективного доступа к исходному коду и исполняемым файлам и осуществление автоматических сборок. Мы обсудим эти проблемы во второй части этой главы.

6.1 Рекурсивный *make*

Мотивация использования рекурсивного *make* довольно проста: *make* отлично работает в рамках одного каталога (или небольшого набора каталогов), однако его использование заметно усложняется с ростом числа каталогов. Таким образом, мы можем использовать *make* для сборки большого проекта, написав для каждого каталога свой простой самодостаточный *makefile*, а затем выполнив полученные *makefile*'ы по очереди. Мы могли бы использовать для этой цели сценарий, однако наиболее эффективным подходом является использование *make*, поскольку между компонентами обычно существуют зависимости более высокого уровня.

Предположим, что мы ведём разработку приложения для воспроизведения mp3 файлов. Логически его можно разделить на несколько компонентов: пользовательский интерфейс, кодеки и система управления базой данных. Эти компоненты могут быть представлены трёмя библиотеками: *libui.a*, *libcodec.a* и *libdb.a*. Само приложение является «клеем», связывающим эти три части воедино. Наиболее простое отображение этих компонентов в структуру каталогов представлено на рисунке 6.1.

Более традиционная структура каталогов подразумевает помещение функции

main и «клея» в корневой каталог, а не в подкаталог *app/player*. Я предпочитаю помещать код приложения в собсвенный каталог, поскольку это делает структуру корневого каталога более ясной и позволяет легко добавлять в систему новые модули. Например, если мы решим добавить отдельное приложение для управления музыкальными каталогами, мы можем аккуратно поместить его в *app/catalog*.

Если каждый каталог из *lib/db*, *lib/codec*, *lib/ui* и *app/player* содержит собственный *makefile*, то работой головного *makefile*'а станет их последовательный вызов:

```
lib_codec := lib/codec
lib_db := lib/db
lib_ui := lib/ui
libraries := $(lib_ui) $(lib_db) $(lib_codec)
player := app/player

.PHONY: all $(player) $(libraries)
all: $(player)

$(player) $(libraries):
    $(MAKE) --directory=$@

$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

Главный *makefile* вызывает *make* в каждом подкаталоге в рамках правила, перечисляющего все подкаталоги как цели и выполняющего вызов *make*:

```
$(player) $(libraries):
    $(MAKE) --directory=$@
```

Переменная `MAKE` должна использоваться всегда при вызове *make* из *makefile*'а. *make* распознаёт эту переменную и подставляет на её место реальный путь к исполняемому файлу *make*, чтобы все рекурсивные вызовы *make* использовали один исполняемый файл. К тому же, строки, содержащие переменную `MAKE`, обрабатываются особым образом, если используются опции `--touch` (`-t`), `--just-print` (`-n`) или `--question` (`-q`). Мы обсудим эти детали в разделе «Опции командной строки» далее в этой главе.

Целевые каталоги помечены как `.PHONY`, поэтому правила выполняются даже в том случае, когда пересборка целей не требуется. Опция `--directory` (`-C`) используется для того, чтобы заставить *make* поменять текущий каталог перед чтением *makefile*'а.

Это правило, также довольно тонкое, помогает избежать нескольких проблем, возникающих при использовании более «очевидного» командного сценария:

```
all:
    for d in $(player) $(libraries); \
```

```
do                                  \
    $(MAKE) --directory=$$d;        \
done
```

Этот командный сценарий не сможет правильно передать ошибки родительскому *make*. Он также не позволит *make* выполнять сборки в разных подкаталогах параллельно. Мы обсудим эту возможность *make* в главе 10.

Когда *make* планирует выполнение графа зависимостей, реквизиты цели выглядят для него независимыми. В добавок к этому, две цели, не связанные зависимостью от третьей цели, также независимы. Например, библиотеки не имеют непосредственной зависимости от цели `app/player` или друг от друга. Это позволяет *make* выполнять *makefile* для *app/player* перед сборкой библиотек. Естественно, это вызовет ошибку сборки, так как компоновка приложения требует наличия библиотек. Для решения этой проблемы мы добавили дополнительную информацию о зависимостях:

```
$(player): $(libraries)
$(lib_ui): $(lib_db) $(lib_codec)
```

Этот отрывок сообщает *make*, что *makefile*'ы в подкаталогах библиотек должны быть выполнены раньше *makefile*'а в каталоге *player*. Точно так же код библиотеки *lib/ui* требует, чтобы библиотеки *lib/db* и *lib/codec* были уже собраны. Это позволяет быть уверенным, что любой код, требующий генерации (например, исходные файлы на yacc/lex), будет сгенерирован до того, как начнётся компиляция кода *ui*.

Существует также одна тонкость в отношении порядка сборки реквизитов. Как и в случае остальных зависимостей, порядок сборки определяется на основании анализа графа зависимостей, однако когда реквизиты цели перечисляются в одной строке, GNU *make* иногда собирает их слева направо. Рассмотрим пример:

```
all: a b c
all: d e f
```

Если нет других зависимостей, требующих рассмотрения, шесть реквизитов могут быть собраны в любом порядке (например, «d b a c e f»), однако *make* в рамках одной строки использует порядок слева направо, порождая один из следующих результатов: «a b c d e f» *или* «d e f a b c». Несмотря на то, что этот порядок является случайностью реализации, порядок выполнения будет выглядеть правильным. Легко забыть, что правильный порядок сборки является счастливой случайностью, и не предоставить *make* полную информацию о зависимостях компонентов. Рано или поздно анализ зависимостей породит другой порядок сборки, став причиной ошибок. Таким образом, если набор целей должен быть собран в

определённом порядке, укажите этот порядок явно при помощи соответствующих реквизитов.

Когда будет выполнен головной *makefile*, мы увидим следующий вывод:

```
make --directory=lib/db
make[1]: Entering directory '/test/lib/db'
Update db library...
make[1]: Leaving directory '/test/lib/db'
make --directory=lib/codec
make[1]: Entering directory '/test/lib/codec'
Update codec library...
make[1]: Leaving directory '/test/lib/codec'
make --directory=lib/ui
make[1]: Entering directory '/test/lib/ui'
Update ui library...
make[1]: Leaving directory '/test/lib/ui'
make --directory=app/player
make[1]: Entering directory '/test/app/player'
Update player application...
make[1]: Leaving directory '/test/app/player'
```

Когда *make* определяет, что происходит рекурсивный вызов *make*, он автоматически включает опцию `--print-directory` (`-w`), руководствуясь которой, *make* печатает сообщения при входе в каталог или выходе из него. Эта опция также автоматически включается при использовании опции `--directory` (`-C`). В добавок ко всему на каждой строке в квадратных скобках печатается значение переменной `MAKELEVEL`. В нашем простом примере *makefile* каждого компонента печатает только сообщение о сборке соответствующего компонента.

6.1.1 Опции командной строки

Рекурсивный *make* — это простая идея, которая очень быстро становится сложной. Идеальная реализация рекурсивного *make* ведёт себя так, будто множество *makefile*'ов системы является одним целым. Такой уровень координации практически не достижим, поэтому в реальности всегда приходится идти на компромисы. Тонкие проблемы станут яснее, когда мы рассмотрим, как должны обрабатываться опции командной строки.

Предположим, что мы добавили комментарии в заголовочный файл нашего mp3 плеера. Вместо перекомпиляции всего исходного кода, зависящего от модифицированного заголовочного файла, мы можем выполнить команду `make --touch`, чтобы обновить время модификации всех файлов. Выполнив эту команду в каталоге с главным *makefile*'ом, мы хотели бы, чтобы *make* обновил временные метки всех файлов, управляемыми дочерними экземплярами *make*. Посмотрим, как это работает.

Когда используется опция `--touch`, обычно нормальный процесс выполнения правил отменяется. Вместо этого *make* производит обход графа зависимостей, обновляя дату модификацию всех запрошенных неабстракных целей и их реквизитов при помощи команды *touch*. Поскольку все наши каталоги помечены как `.PHONY`, при нормальном ходе событий они будут проигнорированы (поскольку обновление даты модификации для них смысла не имеет). Однако мы не хотим, чтобы эти цели игнорировались, нам требуется выполнение ассоциированных с ними правил. Чтобы обеспечить правильное поведение, *make* автоматически помечает все строки в сценариях, содержащие переменную `MAKE`, модификатором `+`, в результате чего *make* запускает дочерние процессы *make*, несмотря на опцию `--touch`.

Когда *make* запускает дочерние процессы *make*, он должен позаботиться о передаче им флага `--touch`. Это достигается при помощи переменной `MAKEFLAGS`. Когда *make* стартует, происходит автоматическое добавление большей части опций к переменной `MAKEFLAGS`. Исключениями являются опции `--directory` (`-C`), `--file` (`-f`), `--old-file` (`-o`) и `--new-file` (`-w`). Переменная `MAKEFLAGS` экспортируется в окружение и считывается дочерними процессами *make* при старте.

Благодаря этой функцональности дочерние процессы *make* по большей части ведут себя так, как вы ожидаете. Рекурсивное выполнение `$(MAKE)` и специальная обработка переменной `MAKEFLAGS`, применяемая к опции `--touch`, также применяется к опциям `--just-print` (`-n`) и `--question` (`-q`).

6.1.2 Передача переменных

Как уже было замечено, переменные передаются в дочерние процессы *make* через окружение и контролируются при помощи директив `export` и `unexport`. Значения переменных, переданные через окружение, принимаются как значения по умалчанию, однако любое присваивание изменит их значение. Для того, чтобы разрешить переменным окружения переопределять локальные присваивания, используйте опцию `--environment-overrides` (`-e`). Вы можете явно переопределить переменную окружения (даже при включённой опции `--environment-overrides`) при помощи директивы `override`:

```
override TMPDIR = ~/tmp
```

Переменные, определённые в командной строке, автоматически экспортируются в окружение, если их имена удовлетворяют синтаксису командного интерпретатора, то есть содержат только буквы, цифры и подчёркивания. Присваивания переменных в командной строке сохраняются в переменной `MAKEFLAGS` наряду с другими опциями.

6.1.3 Обработка ошибок

Что происходит, когда рекурсивный вызов *make* обнаруживает ошибку? На самом деле ничего особенного. Процесс *make*, обнаруживший ошибку, завершается с кодом возврата 2. После этого происходит выход из родительского процесса *make*, и ошибка передаётся вверх по дереву рекурсивных вызовов. Если первый вызов *make* содержал опцию `--keep-going` (`-k`), она передаётся в дочерние процессы. В этом случае дочерний процесс *make* продолжает нормальное выполнение, отбрасывает текущую цель и переходит к следующей, не используя цель, вызвавшую ошибку, в качестве реквизита.

Например, если во время сборки нашего mp3 плеера обнаружится ошибка компиляции в компоненте *lib/db*, *make* закончит выполнение, вернув код ошибки 2 родительскому процессу. Если мы использовали опцию `--keep-going` (`-k`), главный процесс *make* начнёт обработку следующей независимой цели, *lib/codec*. Когда сборка этой цели будет закончена, *make* завершит выполнение с кодом возврата 2, поскольку сборка остальных целей не может быть осуществлена по причине ошибки в *lib/db*.

Опция `--question` (`-q`) приводит к похожему поведению. При включении этой опции *make* возвращает код ошибки 1 в случае, если какая-то цель требует повторной сборки, и 0 в противном случае. Если применить эту опцию к дереву *makefile*'ов, *make* будет рекурсивно выполнять *makefile*'ы, пока не определит, требует ли проект сборки. Как только обнаружится файл, требующий сборки, *make* завершит выполняемый в данный момент процесс *make* и «размотает» рекурсию.

6.1.4 Сборка других целей

Базовые цели для сборки естественны для большинства систем сборок, однако нам нужны и другие вспомогательные цели, от которых мы зависим, такие как `clean`, `install`, `print` и так далее. Поскольку это абстрактные цели, описанная выше техника работает не очень хорошо.

Например, ниже представлены несколько неработающих подходов:

```
clean: $(player) $(libraries)
    $(MAKE) --directory=$@ clean
```

или:

```
$(player) $(libraries):
    $(MAKE) --directory=$@ clean
```

Первый пример не работает потому, что реквизиты цели `clean` вызовут сборку целей по умолчанию в *makefile*'ах `$(player)` и `$(libraries)`, а не сборку цели `clean`. Второй пример неверен потому, что для этих целей уже определён другой командный сценарий.

Один из рабочих подходов основывается на использование цикла `for`:

```
clean:
    for d in $(player) $(libraries); \
    do                               \
      $(MAKE) --directory=$$f clean; \
    done
```

Цикл `for` не очень хорошо отвечает всем доводам, приведённым ранее, однако он (вместе с предыдущим неверным примером) приводит нас к следующему решению:

```
$(player) $(libraries):
    $(MAKE) --directory=$@ $(TARGET)
```

Добавив к строке с рекурсивным вызовом *make* переменную `TARGET` и выставляя значение этой переменной через командную стоку, мы можем собирать в дочерних процессах *make* произвольные цели:

```
$ make TARGET=clean
```

К сожалению, это не приведёт к сборке цели `$(TARGET)` в головном *makefile*'е. Часто это неважно, поскольку головной *makefile* не делает ничего, однако в случае необходимости мы можем добавить ещё один вызов *make*, защищённый функцией *if*:

```
$(player) $(libraries):
    $(MAKE) --directory=$@ $(TARGET)
    $(if $(TARGET), $(MAKE) $(TARGET))
```

Теперь мы можем собрать цель `clean` (или любую другую), просто присвоив соответствующее значение переменной `TARGET` в командной строке.

6.1.5 Общие зависимости

Специальная поддержка *make* переменных командной строки и коммуникация через переменные окружения подразумевают, что механизм рекурсивного *make* хорошо отлажен. Так в чём же заключаются упомянутые ранее сложности?

Разделённые *makefile*'ы, соединяемые воедино командами `$(MAKE)` описывают только наиболее поверхностные высокоуровневые связи. К сожалению, часто бывают более тонкие зависимости, скрытые в некоторых каталогах.

Предположим для примера, что модуль *db* включает анализатор, основанный на *yacc*, для импорта и экспорта мызукальных данных. Если модуль *ui*, *ui.c*, включает сгенерированный *yacc* заголовочный файл, на лицо связи между этими двумя модулями. Если зависимости смоделированы правильно, *make* должен знать, что модуль *ui* требует пересборки в случае изменения заголовочного файла грамматики. Это нетрудно организовать, используя технику автоматической генерации

зависимостей, описанную ранее. Однако что если исполняемый файл *yacc* также изменился? В этом случае после запуска *makefile*'а модуля *ui* корректный *makefile* определит, что сначала должна быть выполнена команда *yacc* для генерации анализатора и заголовочного файла, и только после этого должна быть осуществлена компиляция *ui.c*. При нашей декомпозиции этого не случится, потому что правила для запуска *yacc* находятся в *makefile*'е *db*, а не *ui*.

В этом случае лучшее, что мы можем сделать — это убедиться в том, что *makefile* модуля *db* запускается всегда раньше *makefile*'а модуля *ui*. Эта высокоуровневая зависимость должна быть указана вручную. Мы были достаточно проницательны, чтобы указать эту зависимость в первой версии нашего *makefile*'а, однако в целом это может стать серьёзной проблемой при поддержке. Поскольку код добавляется и модифицируется, головной *makefile* в какой-то момент будет неправильно описывать зависимости между модулями.

В продолжение примера предположим, что грамматика *yacc* в модуле *db* была изменена, и *makefile* модуля *ui* был выполнен до *makefile*'а модуля *db* (вручную в обход головного *makefile*'а). *makefile* модуля *ui* не содержит информации о неудовлетворённой зависимости в *makefile*'е модуля *db* и о необходимости запуска программы *yacc* для изменения головного файла. Вместо этого *makefile* модуля *ui* компилирует программу с устаревшим заголовочным файлом. Если при модификации были добавлены новые символы, будет обнаружена ошибка компиляции. Поэтому рекурсивный подход изначально более хрупок по сравнению с монолитным *makefile*'ом.

Ситуация ухудшается с повышением интенсивности использования генераторов исходного кода. Предположим, в реализации модуля *ui* был использован генератор заглушек RPC[2], заголовочные файлы которых используются в модуле *db*. Теперь нам придётся бороться с перекрёстными ссылками между модулями. Для решения этой проблемы нам придётся сначала посетить модуль *db* и сгенерировать заголовочные файлы *yacc*, затем посетить модуль *ui* и сгенерировать заглушки RPC, затем вернуться в *db* и произвести компиляцию, и, наконец, посетить *ui* и завершить процесс компиляции. Число проходов, требуемое для создания и компиляции исходного кода проекта зависит от структуры кода и инструментов, при помощи которых он создаётся. Такой вид перекрёстных зависимостей встречается в сложных системах довольно часто.

Стандартные решения в настоящих *makefile*'ах как правило являются уловками. Для того, чтобы убедиться, что обновлены все файлы, каждый *makefile* выполняется по команде головного *makefile*'а. Заметьте, что это именно тот подход, который мы использовали в примере с mp3 плеером. Когда происходит запуск головного *makefile*'а, каждый из четырёх дочерних *makefile*'ов запускается по очереди.

[2]RPC (remote procedure call, вызов удалённых процедур) — класс технологий, позволяющий программному обеспечению вызывать функции, находящиеся в другом адресном пространстве (прим. переводчика).

В более сложных случаях для проверки того, что весь код сначала сгенерирован, и только затем скомпилирован, дочерние *makefile*'ы запускаются по несколько раз. Чаще всего такие итерации являются напрасной тратой времени, однако иногда они действительно необходимы.

6.1.6 Избегаем дублирования кода

Структура каталогов нашего приложения включает три библиотеки. *makefile*'ы этих библиотек очень похожи. Несмотря на то, что эти библиотеки служат разным целям, все они собираются похожими командами. Этот тип декомпозиции типичен для больших проектов и ведёт к большому количеству похожих *makefile*'ов и дублированию сценариев.

Дублирование кода — это плохо, даже если оно происходит в *makefile*'е. Оно увеличивает стоимость поддержки программного обеспечения и ведёт к росту количества ошибок. Оно также затрудняет понимание алгоритмов и определение небольших их вариаций. Поэтому желательно избежать дублирования кода *makefile*'ов, настолько это возможно. Легче всего достигнуть этого выносом общих частей в отдельный включаемый файл.

Например, *makefile* модуля *codec* содержит следущее:

```
lib_codec := libcodec.a
sources := codec.c
objects := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))

include_dirs := .. ../../include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

all: $(lib_codec)

$(lib_codec): $(objects)
    $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
    $(RM) $(lib_codec) $(objects) $(dependencies)

ifneq "$(MAKECMDGOALS)" "clean"
  include $(dependencies)
endif

%.d: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
    sed 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
    mv $@.tmp $@
```

Почти весь этот код дублицируется в *makefile*'ах модулей *db* и *ui*. Единственное, что изменяется — это имя библиотеки и исходные файлы. После того, как весь дублированный код вынесен в файл *common.mk*, мы можем сократить предыдущий *makefile* следующим образом:

```
library := libcodec.a
sources := codec.c

include ../../common.mk
```

Посмотрим, что вынесено в единственный общий включаемый файл:

```
MV := mv -f
RM := rm -f
SED := sed

objects      := $(subst .c,.o,$(sources))
dependencies := $(subst .c,.d,$(sources))
include_dirs := .. ../../include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))

vpath %.h $(include_dirs)

.PHONY: library
library: $(library)

$(library): $(objects)
    $(AR) $(ARFLAGS) $@ $^

.PHONY: clean
clean:
    $(RM) $(objects) $(program) $(library) \
          $(dependencies) $(extra_clean)

ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

%.c %.h: %.y
    $(YACC.y) --defines $<
    $(MV) y.tab.c $*.c
    $(MV) y.tab.h $*.h

%.d: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
    $(SED) 's,\($*\.o\) *:,\1 $@: ,' > $@.tmp
    $(MV) $@.tmp $@
```

Переменная `include_dirs`, которая раньше была разной для каждого *makefile*'а, теперь одинакова во всех *makefile*'ах. Это достигнуто благодаря переработке

пути, используемого для поиска заголовочных файлов при компиляции: теперь все библиотеки используют один и тот же путь.

Файл *common.mk* включает также цель по умолчанию для файлов библиотек. Исходные *makefile*'ы использовали в качестве цели по умолчанию `all`. Это вызвало бы проблемы в *makefile*'ах программ, которым требуется указать различные наборы реквизитов для своих целей по умолчанию. Поэтому включаемая версия кода использует цель по умолчанию `library`.

Заметим, что поскольку общий файл содержит цели, в *makefile*'ы программ он должен включаться *после* цели по умолчанию. Заметим также, что команда сценария `clean` содержит ссылки на переменные `program`, `library` и `extra_clean`. Для *makefile*'ов библиотек переменная `program` содержит пустую строку, в *makefile*'ах программ пустую строку содержит переменная `library`. Переменная `extra_clean` добавлена специально для *makefile*'а модуля *db*. Этот *makefile* использует переменную для обозначения кода, сгенерированного программой *yacc*. Код *makefile*'а представлен ниже:

```
library := libdb.a
sources := scanner.c playlist.c
extra_clean := $(sources) playlist.h

.SECONDARY: playlist.c playlist.h scanner.c

include ../../common.mk
```

При использовании этой техники дублирование кода может быть сведено к минимуму. Поскольку бóльшая часть кода вынесена во включаемый *makefile*, со временем он эволюционирует в общий *makefile* всего проекта. Для настройки используются переменные *make* и функции, определяемые пользователем, позволяющие модифицировать общий *makefile* для каждого конкретного каталога.

6.2 Нерекурсивный *make*

Проекты, содержащие множество каталогов, могут управляться и без рекурсивного *make*. Разница заключается в том, что исходные файлы, которыми манипулирует *makefile*, находятся более чем в одном каталоге. Чтобы отразить этот факт, все ссылки на файлы должны использовать абсолютные или относительные пути к файлам.

Часто *makefile* большого проекта содержит множество целей, по одной для каждого модуля системы. Например, в нашем проекте mp3 плеера нам понадобились цели для каждой библиотеки и каждого приложения. Также полезным может быть включение абстрактных целей для групп компонентов, таких, например, как группа всех библиотек. Цель по умолчанию, как правило, производит сборку всех этих

целей. Часто цель по умолчанию также производит составление документации и запуск процедур автоматического тестирования.

Наиболее простой способ использования нерекурсивного *make* — включение всех целей, ссылок на объектные файлы и зависимостей в один *makefile*. Это часто не устраивает разработчиков, знакомых с рекурсивным *make*, поскольку в этом случае вся информация о файлах и каталогах сосредоточена в одном файле, в то время как сами файлы рассредоточены по файловой системе. Для решения этой проблемы Миллер в своей статье о нерекурсивном *make* предлагает добавлять в каждый каталог включаемый файл, содержащий список файлов модуля и правила, специфичные для него. Головной *makefile* включает все дочерние *makefile*'ы.

Следующий пример демонстрирует *makefile* нашего проекта mp3 плеера, включающий *makefile*'ы модулей из соответствующих каталогов.

```
# Информация о каждом модуле хранится в следующих четырёх
# переменных. Инициализируем их как простые переменные.
programs    :=
sources     :=
libraries   :=
extra_clean :=

objects      = $(subst .c,.o,$(sources))
dependencies = $(subst .c,.d,$(sources))

include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include lib/codec/module.mk
include lib/db/module.mk
include lib/ui/module.mk
include app/player/module.mk

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
    $(RM) $(objects) $(programs) $(libraries) \
          $(dependencies) $(extra_clean)
```

```
  ifneq "$(MAKECMDGOALS)" "clean"
    include $(dependencies)
  endif

%.c %.h: %.y
    $(YACC.y) --defines $<
    $(MV) y.tab.c $*.c
    $(MV) y.tab.h $*.h

%.d: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
    $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
    $(MV) $@.tmp $@
```

Далее приведён пример *makefile*'а модуля */lib/codec* (*module.mk*):

```
local_dir  := lib/codec
local_lib  := $(local_dir)/libcodec.a
local_src  := $(addprefix $(local_dir)/,codec.c)
local_objs := $(subst .c,.o,$(local_src))

libraries += $(local_lib)
sources   += $(local_src)

$(local_lib): $(local_objs)
    $(AR) $(ARFLAGS) $@ $^
```

Таким образом, информация, специфичная для модуля, хранится во включаемом файле в каталоге соответствующего модуля. Головной *makefile* содержит только список модулей и директивы `include`. Давайте рассмотрим файл *module.mk* более детально.

Каждый файл *module.mk* добавляет к переменной `libraries` имя текущей библиотеки, а к переменной `sources` — пути к исходным файлам. Переменные с префиксом `local_` используются для хранения констант или для предотвращения повторного вычисления значений. Обратите внимание на то, что каждый модуль использует одинаковые имена `local_` переменных. Именно поэтому вместо рекурсивных переменных используются простые (объявляемые при помощи оператора `:=`): так сборки, затрагивающие несколько *makefile*'ов, не подвержены риску повреждения значений переменных в отдельных *makefile*'ах. Как уже упоминалось, имена библиотек и списки исходных файлов используют относительные пути. Наконец, включаемый файл содержит правила для сборки текущей библиотеки. Использование `local_` переменных в правилах вполне допустимо, так как цели и реквизиты правил вычисляются при чтении файла.

Первые четыре строки головного *makefile*'а определяют переменные, дополняемые информацией о каждом отдельном модуле. Эти переменные должны быть простыми, поскольку каждый модуль будет добавлять к ним данные из локальных переменных:

```
local_src := $(addprefix $(local_dir)/,codec.c)
...
sources   += $(local_src)
```

Если бы переменная `sources` была рекурсивной, финальное её значение содержало бы просто последнее значение `local_src`, повторяющееся снова и снова. Поскольку по умолчанию переменные являются рекурсивными, применяется явная инициализация пустым значением.

Следующий раздел содержит вычисление списка объектных файлов, `objects`, и списка файлов зависимостей при помощи значения переменной `sources`. Эти переменные являются рекурсивными, поскольку на данном этапе обработки *makefile*'а переменная `sources` содержит пустое значение. Это значение не будет использоваться до тех пор, пока не будут прочитаны включаемые *makefile*'ы. В нашем случае наиболее разумно было бы поместить определение этих переменных после директив включения и объявить эти переменные как простые, однако расположение переменных, хранящих списки файлов (`sources`, `libraries`, `objects`), рядом друг с другом упрощает понимание *makefile*'а в целом и является хорошей практикой. К тому же, в более сложных ситуациях перекрёстные ссылки между переменными потребовали бы использования рекурсивных переменных.

Далее мы специфицируем обработку заголовочных файлов C, указывая значение переменной `CPPFLAGS`. Это позволяет компилятору находить заголовочные файлы. Для этой цели используется дополнение значения (оператор `+=`), поскольку заранее нельзя сказать, что значение переменной не определено: опции командной строки, переменные окружения или конструкции *make* могли уже придать ей какое-то значение. Директива `vpath` позволяет *make* находить заголовочные файлы, располагающиеся в других каталогах. Переменная `include_dirs` используется для того, чтобы избежать повторного вычисления списка включаемых каталогов.

Переменные `MV`, `RM` и `SED` используются для того, чтобы избежать жёсткой привязки к конкретным программам. Обратите внимание на регистр имён переменных. Здесь мы следовали соглашениям, принятым в руководству по *make*. Имена переменных, используемых только внутри *makefile*'а, состоят из прописных букв, имена переменных, значение которых можно задать из командной строки — из заглавных.

В следующей секции *makefile*'а всё ещё интереснее. Мы начинаем раздел явных правил с указания цели по умолчанию, `all`. К сожалению, реквизитом цели `all` является переменная `programs`. Эта переменная будет вычислена незамедлительно, однако её значение будет известно только после чтения включаемых файлов. Таким образом, нам требуется прочитать включаемые файлы перед тем, как определить цель `all`. Однако включаемые файлы содержат цели, первая из которых станет целью по умолчанию. Чтобы разрешить эту дилему, мы можем указать цель `all` без реквизитов, прочитать включаемые файлы и добавить реквизиты к цели `all` позднее.

Оставшаяся часть *makefile*'а уже знакома вам по предыдущим примерам, однако всё же стоит обратить внимание на то, как *make* применяет неявные правила. Исходные файлы располагаются в подкаталогах. Когда *make* пытается применить стандартное правило `%.o: %.c`, реквизитом будет файл с относительным путём, например, *lib/ui/ui.c*. *make* автоматически распространит относительный путь на файл цели и попробует собрать *lib/ui/ui.o*. Таким образом, *make* автомагически (automagically) делает именно то, что нужно.

Есть один неприятный сбой. Несмотря на то, что *make* обрабатывает пути должным образом, не все инструменты, используемые им, делают тоже самое. В частности, при использовании *gcc* для генерации зависимостей, результирующий файл не будет содержать относительного пути к целевому объектному файлу. Вывод команды `gcc -M` будет следующим:

```
ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

в то время как мы ожидаем увидеть другое:

```
lib/ui/ui.o: lib/ui/ui.c include/ui/ui.h lib/db/playlist.h
```

Это нарушает обработку файлов реквизитов. Для решения этой проблемы мы можем настроить команду *sed* так, чтобы она добавляла информацию об относительных путях:

```
$(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,'
```

Тонкая настройка *makefile*'а для обхода причуд различных инструментов является естественной частью работы с *make*. Код переносимых *makefile*'ов часто бывают очень сложным из-за капризов различных наборов инструментов, на которые приходится полагаться.

Теперь у нас есть добротный нерекурсивный *makefile*, однако при поддержке могут возникнуть проблемы. Дело в том, что включаемые файлы *module.mk* во многом схожи. Изменения в одном из них скорее всего приведут к необходимости менять другие файлы. Для небольшого проекта наподобие нашего mp3 плеера это неприятно. Для большого проекта, содержащего несколько сотен включаемых файлов, это может быть фатально. При разумном выборе имён переменных и регуляризации содержимого включаемых файлов эта болезнь поддаётся лечению. Ниже приводится включаемый файл *lib/codec* после рефакторинга:

```
local_src := $(wildcard $(subdirectory)/*.c)
$(eval $(call make-library,
        $(subdirectory)/libcodec.a,
 $(local_src)))
```

Вместо того, чтобы перечислять исходные файлы явно, мы используем предположение, согласно которому в сборке нуждаются все исходные файлы в каталоге. Функция *make-library* осуществляет набор операций, общих для всех включаемых файлов. Эта функция определяется в начале головного *makefile*'а нашего проекта:

```
# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
    $(AR) $(ARFLAGS) $$@ $$^
endef
```

Функция добавляет исходные файлы и имя библиотеки к соответствующим переменным, затем определяет явные правила для сборки библиотеки. Обратите внимание на то, что автоматические переменные используются с двумя знаками доллара, чтобы отложить их вычисление до выполнения правила. Функция *source-to-object* трансформирует список исходных файлов в список соответствующих объектных файлов:

```
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
```

В предыдущей версии *makefile*'а мы затушевали тот факт, что настоящими исходными файлами являются *playlist.y* и *scanner.l*. Вместо этого в списке файлов мы указывали сгенерированные *.c* файлы. Из-за этого нам приходилось указывать их явно и включать дополнительную переменную, `extra_clean`. Мы решили эту проблему, позволив переменной `sources` содержать файлы *.y* и *.l* и возложив на функцию *source-to-object* работу по переводу имён этих файлов в имена соответствующих объектных файлов.

В дополнение к модификации функции *source-to-object* нам нужно добавить ещё одну функцию, вычисляющую имена выходных файлов *yacc* и *lex*, чтобы позволить цели `clean` должным образом выполнять свою работу. Функция *generated-source* принимает на вход список файлов и возвращает список промежуточных файлов:

```
# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1)) \
                   $(subst .y,.h,$(filter %.y,$1)) \
                   $(subst .l,.c,$(filter %.l,$1))
```

Другая полезная функция, *subdirectory*, помогает избавиться от локальной переменной `local_dir`.

```
subdirectory = $(patsubst %/makefile,%, \
                 $(word                 \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

Как уже было замечено в разделе «Строковые функции» главы 4, мы можем получить имя текущего *makefile*'а из переменной `MAKEFILE_LIST`. Использовав функцию *patsubst*, мы можем извлечь относительный путь из имени последнего прочитанного *makefile*'а. Это помогает устранить одну переменную и уменьшить разницу между включаемыми файлами.

Нашей последней оптимизацией (по крайней мере, в этом примере) является использование функции *wildcard* для получения списка исходных файлов. Это прекрасно работает в большинстве сред, поддерживающих чистоту в каталогах с исходными файлами. Однако мне приходилось работать в проекте, в котором это было не принято. Старый код хранился в каталогах с исходными файлами «на всякий случай». Это влекло реальные затраты, выраженные во времени и нервах программистов, поскольку средства поиска и замены находили символы в старом коде, и новые программисты (или старые, не знакомые с модулем) пытались откомпилировать и отладить код, который никогда не использовался. Если вы используете современную систему контроля версий (например, CVS), хранение старого кода в каталогах с исходным кодом совершенно бессмысленно (поскольку весь код уже хранится в репозитории), и использование *wildcard* становится оправданным.

Директивы `include` также могут быть оптимизированы:

```
modules := lib/codec lib/db lib/ui app/player
...
include $(addsuffix /module.mk,$(modules))
```

Для больших проектов даже этот код может стать проблемой при поддержке, поскольку список модулей может вырасти до сотен или даже тысяч. При некоторых обстоятельствах более предпочтительным является автоматическое определение модулей при помощи команды *find*:

```
modules := $(subst /module.mk,,
             $(shell find . -name module.mk))
...
include $(addsuffix /module.mk,$(modules))
```

Мы обрезаем имена файлов, обнаруженных командой *find*, делая переменную `modules` более полезной как список модулей. Если вам этого не требуется, тогда, конечно, можно опустить вызовы *subst* и *addsuffix* и просто сохранить вывод команды *find* в переменной `modules`. Следующий пример демонстрирует результирующий *makefile*.

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))

# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%, \
                 $(word                  \
                   $(words $(MAKEFILE_LIST)),
           $(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $1
  sources   += $2
  $1: $(call source-to-object,$2)
      $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(subst .y,.c,$(filter %.y,$1)) \
                   $(subst .y,.h,$(filter %.y,$1)) \
                   $(subst .l,.c,$(filter %.l,$1))

# Информация о каждом модуле хранится в следующих четырёх
# переменных. Инициализируем их как простые переменные.
modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources      :=
objects      = $(call source-to-object,$(sources))
dependencies = $(subst .o,.d,$(objects))

include_dirs := lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MV  := mv -f
RM  := rm -f
SED := sed

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)
```

```
.PHONY: clean
clean:
    $(RM) $(objects) $(programs) $(libraries) $(dependencies) \
          $(call generated-source, $(sources))

  ifneq "$(MAKECMDGOALS)" "clean"
    include $(dependencies)
  endif

%.c %.h: %.y
    $(YACC.y) --defines $<
    $(MV) y.tab.c $*.c
    $(MV) y.tab.h $*.h

%.d: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $< | \
    $(SED) 's,\($(notdir $*)\.o\) *:,$(dir $@)\1 $@: ,' > $@.tmp
    $(MV) $@.tmp $@
```

Использование одного включаемого файла для каждого модуля является весьма работоспособным подходом и имеет свои преимущества, однако я не могу с уверенностью сказать, что он является наилучшим. Мой собственный опыт работы с проектом на Java показывает, что использование головного *makefile*'а, эффективно включающего файлы модулей, является разумным решением. Этот проект включал 997 отдельных модулей, около двух десятков библиотек и полдюжины приложений. Для обработки несвязанных подмножеств кода использовались различные *makefile*'ы. Эти файлы в совокупности содержали примерно 2500 строк. Общий включаемый файл, содержащий глобальные переменные, функции, определяемые пользователем, и шаблонные правила, содержал ещё примерно 2500 строк.

Выберите ли вы один единственный *makefile*, или же поместите информацию о модулях в отдельные включаемые файлы, нерекурсивный *make* является хорошим подходом к сборке крупных проектов. Он также решает много традиционных проблем, связанных с использованием рекурсивного *make*. Единственный недостаток этого подхода, о котором стоит предупредить – для разработчиков, использовавших рекурсивный *make*, потребуется смена парадигмы.

6.3 Компоненты больших систем

Мы рассмотрим две популярных на сегодняшний день модели разработки: модель свободного программного обеспечения и коммерческую модель.

В модели свободного программного обеспечения каждый разработчик в основном может полагаться только на себя. Проект имеет *makefile* и *README* файл, и ожидается, что разработчикам потребуется лишь немного помощи для начала

работы. Приоритетами таких проектов, как правило, являются качество и привлечение к участию всего сообщества, однако наибольшую ценность имеет участие умелых и хорошо мотивированных членов сообщества. Это не критика. С этой точки зрения программное обеспечение должно быть высокого качества, соблюдение временных ограничений имеет меньший приоритет.

В коммерческой модели разработки разработчики могут иметь различный уровень подготовки, и все из них должны быть способны выполнить работу к назначенному сроку. Разработчик, который не может разобраться, как выполнить свою работу, расходует деньги понапрасну. Если система не компилируется или не запускается должным образом, вся команда разработчиков может простаивать: этот сценарий является наиболее затратным из всех возможных. Для решения подобных проблем процесс разработки управляется командой поддержки, координирующей процесс сборки, конфигурацию инструментов разработки, поддержку и разработку, а также менеджмент новых релизов. В подобной среде процессом управляют критерии эффективности.

Как правило, для коммерческой модели характерны более продуманные системы сборки. Основной причиной такого перевеса является стремление сократить стоимость разработки программного обеспечения за счёт повышения эффективности труда программистов. Это, в свою очередь, должно вести к увеличению прибыли. Это именно та модель, которая требует от *make* наибольшего функционала. Тем не менее, техники, которые мы обсудим, применимы в случае необходиости и к модели свободного программного обеспечения.

Этот раздел содержит много общей информации, чуть-чуть специфики и совсем не содержит примеров. Причина этого заключается в том, что очень многое зависит от языка разработки и используемой операционной среды. В главах 8 и 9 мы рассмотрим примеры реализации многих возможностей, рассмотренных в этом разделе.

Требования

Разумеется, требования различны для каждого проекта и каждой рабочей среды. Здесь мы рассмотрим основные требования, считающиеся важными во многих коммерческих средах разработки.

Наиболее общей потребностью команд разработчиков является отделение исходного кода от бинарных файлов. То есть объектные файлы, полученные в результате компиляции, должны располагаться в отдельном дереве каталогов. Это, в свою очередь, позволяет включать ещё множество возможностей. Отделение дерева каталогов бинарных файлов сулит множество преимуществ:

- Когда расположение дерева каталогов бинарных файлов определено, гораздо легче управлять дисковым пространством.

- Различные версии деревьев бинарных файлов могут управляться параллельно. Например, единственному дереву исходных файлов могут соответствовать оптимизированное, отладочное, и профилировочное деревья бинарных файлов.

- Существует возможность одновременной поддержки различных платформ. Правильным образом реализованное дерево исходных файлов может быть использовано для параллельной компиляции исполняемых файлов для различных платформ.

- Разработчики могут взять небольшую часть исходного кода и позволить системе сборки самостоятельно «заполнять» недостающие файлы с помощью зависимостей исходных файлов и деревьев каталогов объектных файлов. Это не обязательно требует отделения исходных файлов от объектных, однако без отделения больше вероятность того, что система сборки не сможет правильно определить, где следует искать бинарные файлы.

- Дерево исходного кода может быть сделано доступно только для чтения. Это даёт дополнительную уверенность в том, что сборка отражает реальное состояние репозитория.

- Некоторые цели, подобные `clean`, можно реализовать тривиальным образом (и выполнять с колоссальным выигрышем в производительности), если всё дерево каталогов может быть рассмотрено как отдельная единица, не требующая поиска и манипуляций файлами.

Бóльшая часть этих пунктов является важными преимуществами системы сборки и может быть проектным требованием.

Возможность управления историей сборок проекта часто является важным качеством системы сборки. Основная идея заключается в том, что сборка исходного кода осуществляется по ночам, обычно при помощи задачи *cron*. Поскольку результирующие деревья каталогов, содержащие исходный код и бинарные файлы, являются не модифицируемыми с точки зрения CVS, я буду называть их справочными. Эта идея имеет множество применений.

Во-первых, справочное дерево каталогов исходного кода может использоваться программистами и менеджерами, которым нужно просмотреть исходный код. Это может показаться банальным, однако когда число файлов и релизов растёт, извлечение всего исходного кода из репозитория ради просмотра одного файла может быть не очень разумно. К тому же, хоть инструменты для просмотра CVS репозиториев достаточно распространены, они обычно не предоставляют средств для простого поиска по всему исходному коду проекта. Для этих целей больше подходят таблицы символов или даже команды *find*/*grep* (или *grep -R*).

Во-вторых, справочные деревья бинарных файлов являются индикатором того, что соответствующая сборка прошла успешно. Когда разработчики начинают утром свою работу, они уже знают, является ли система работоспособной. Если проект использует систему автоматического тестирования, свежие сборки могут использоваться для запуска автоматических тестов. Каждый день разработчики могут проверять отчёты с результатами тестов, чтобы определить жизнеспособность системы, не тратя время на запуск тестов. Если же разработчик имеет на руках только модифицированную версию исходного кода, имеет место дополнительное сокращение затрат проекта, поскольку в этом случает разработчику не нужно терять время на получение исходной версии и сборку. Наконец, справочные сборки могут запускаться разработчиками для тестирования и сравнения функциональности определённых компонентов.

Есть и другие способы использования справочных сборок. Для проекта, состоящего из множества библиотек, прекомпилированные библиотеки, полученные в результате ночных сборок, могут использоваться программистами для компоновки собственных приложений с теми библиотеками, которые они не модифицируют. Это позволяет сократить цикл разработки за счёт исключения необходимости компиляции большей части исходного кода при запуске собственных сборок. Разумеется, лёгкий доступ к исходному коду проекта, располагающегося на локальном файловом сервере, чрезвычайно удобен, если разработчикам, не имеющим полной рабочей копии проекта, нужно просмотреть исходный код.

При таком разнообразии применений справочных деревьев каталогов поддержание целостности их структуры становится чрезвычайно важным. Одним из простых и эффективных способов повышения надёжности является объявление дерева каталогов с исходным кодом доступным только для чтения. Это гарантирует, что дерево отражает состояние репозитория в момент сборки. Этот аспект может потребовать особого внимания, поскольку во многих случаях система сборки может принимать попытки записи в дерево каталогов, в частности, при генерации исходного кода или создании временных файлов. Объявления дерева каталогов с исходным кодом доступным только для чтения также предотвращает случайное его повреждение обычными пользователями, что случается наиболее часто.

Ещё одним общим требованием к проектной системе сборки является возможность лёгкого управления различными конфигурациями компиляции, компоновки и развёртывания системы. Система сборки обычно должна оперировать различными версиями проекта (которые могут быть различными ветками в репозитории).

Множество крупных проектов зависят от программного обеспечения третих разработчиков, представленном в форме библиотек или инструментов разработки. Если нет других инструментов для управления конфигурацией программного обеспечения (а обычно их нет), использование *makefile*'а и системы сборки для этих целей часто является разумным выбором.

Наконец, когда программное обеспечение доставляется заказчику, оно часто

упаковывается на базе текущей рабочей версии. Это может быть также сложно, как конструирование *setup.exe* файла для Windows или также просто, как редактирование HTML файла и связывание его с *jar* архивом. Иногда операция инсталляции сочетается с обычным процессом сборки. Я предпочитаю разделять сборку и генерацию инсталлятора на два независимых шага, поскольку, как правило, они используют совершенно разные процессы. В любом случае, скорее всего, обе эти операции будут влиять на систему сборки.

6.4 Структура файловой системы

Как только вы решите поддерживать несколько деревьев каталогов, содержащих бинарные файлы, встаёт вопрос о структуре файловой системы. В средах, требующих использования нескольких деревьев каталогов, часто содержится *много* таких деревьев. Чтобы найти способ поддержания порядка в таких средах требуется немного подумать.

Наиболее общим способом структурирования этих данных является выделение большого жёсткого диска как хранилища деревьев каталогов с бинарными файлами. Корневой (или близкий к корню) каталог содержит один подкаталог для каждого дерева. Одним из разумных способов идетнификации этих деревьев является включение в имя каждого каталога именования поставщика, платформы, операционной системы и параметров сборки бинарного дерева:

```
$ ls
hp-386-windows-optimized
hp-386-windows-debug
sgi-irix-optimized
sgi-irix-debug
sun-solaris8-profiled
sun-solaris8-debug
```

Если требуется хранить множество сборок, произведённых в разные моменты времени, для их идентификации обычно используются временные метки, включенные в имя каталога. Из-за удобства сортировки часто используются форматы `гг-мм-дд` и `гг-мм-дд-чч-мм`:

```
$ ls
hp-386-windows-optimized-040123
hp-386-windows-debug-040123
sgi-irix-optimzed-040127
sgi-irix-debug-040127
sun-solaris8-profiled-040127
sun-solaris8-debug-040127
```

```
release
|
|--product1
|  |--1.0
|  |  |--040101
|  |  `--040112
|  |
|  `--1.4
|     `--040121
|
`--product1
   `--1.4
      `--031212
```

Рис. 6.2: Пример структуры дерева каталогов релиза

Конечно, способ упорядочивания имён компонентов целиком зависит от ваших потребностей. Каталог верхнего уровня этих деревьев хорошо подходит для хранения *makefile*'а и отчётов о результатах тестов.

Предыдущая структура хорошо подходит для хранения множества сборок, осуществляемых разработчиками параллельно. Если команда разработчиков выпускает «релизы», возможно, для внутренних потребителей, вам стоит рассмотреть возможность добавления хранилища релизов, структурированное как множество продуктов, каждый из которых может иметь номер ревизии и временную метку, как показано на рисунке 6.2.

Продукты могут быть библиотеками, производимыми командой разработчиков для нужд других разработчиков. Конечно, это могут быть и продукты в привычном их понимании.

Каковы бы ни были структура файловой системы и среда разработки, реализацией управляет множество однотипных критериев. Каждое дерево должно легко идентифицироваться. Освобождение ресурсов должно быть быстрым и ясным. Полезно иметь возможность перемещать и архивировать деревья. В добавок ко всему, структура файловой системы должна быть близка структуре процесса разработки организации. Это позволит перемещаться по хранилищу непрограммистам, таким, как менеджеры, инженеры по качеству и составители технической документации.

6.5 Автоматические сборки и тестирование

Как правило важно иметь возможность максимально возможной автоматизации процесса сборки. Это позволит производить сборку справочных деревьев каталогов по ночам, сохраняя дневное время разработчиков. Это также позволяет разработчикам запускать сборки на собственных машинах без предварительной

подготовки.

Для программного обеспечения, находящегося в разработке, часто возникает множество заявок на сборку различных версий различных продуктов. Для человека, выполняющего эти заявки, возможность запланировать несколько сборок и «пойти прогуляться» часто является критичной для поддержки и выполнения заявок.

Автоматизированное тестирование создаёт дополнительные трудности. Для управления процессом тестирования большинства консольных приложений могут быть использованы простые сценарии. Для тестирования консольных приложений, требующих взаимодействия с пользователем, можно использовать утилиту GNU *dejaGnu*. Разумеется, каркасы, подобные JUnit (*http://www.junit.org*), также предоставляют поддержку модульного тестирования приложений, не требующего графической среды.

Тестирование приложений с графическим пользовательским интерфейсом готовит дополнительные проблемы. Для систем, использующих X11, я с успехом применял тестирование по расписанию с использованием виртуального оконного буфера (virtual frame buffer), Xvfb. Для Windows я не смог найти удовлетворительного решения для автоматизированного тестирования. Все подходы основаны на сохранении тестовой учётной записи зарегистрированной системе, а экрана — не заблокированным.

Глава 7

Переносимые *makefile*'ы

Какой же *makefile* мы будем считать переносимым? В качестве экстремального примера мы хотим иметь *makefile*, запускающийся без изменений на любой платформе, позволяющей запустить GNU *make*. Однако это практически невозможно по причине огромного количества различных операционных систем. Более разумным будет назвать переносимым *makefile*, который легко изменить для запуска на другой платформе. Однако перенос на другую платформу не должен препятствовать поддержке всех предыдущих платформ, это будет дополнительным важным ограничением.

Мы можем достичь этого уровня переносимости *makefile*'ов, используя те же техники, что и в традиционном программировании: инкапсуляция и абстракция. Используя переменные и функции, определяемые пользователем, мы можем инкапсулировать приложения и алгоритмы. Определяя переменные для аргументов командной строки и параметров, мы можем абстрагироваться от элементов, варьирующихся от платформы к платформе.

Затем вам потребуется определить, какие инструменты может предложить каждая платформа для выполнения вашей работы, и какие из них нужно использовать в случае каждой конкретной платформы. Наибольшую переносимость приносит использование только тех инструментов, которые присутствуют на всех интересующих платформах. Обычно это называют подходом «наименьшего общего знаменателя», который, очевидно, может сделать базовый набор инструментов довольно скудным.

Другой версией подхода наименьшего общего знаменателя является следующая парадигма: используйте мощный набор инструментов и убедитесь, что можете взять его с собой на любую платформу. Это гарантирует, что команды, которые вы вызываете в *makefile*'е, работают совершенно одинаково в любой системе. Осуществить это, как правило, нелегко, и с административной точки зрения, и в плане убеждения вашей организации в необходимости кооперации её систем с

вашими наработками. Однако такой подход может приносить результаты, и позже я приведу приведу пример с пакетом Cygwin для Windows. Как вы увидите, стандартизация инструментов не решает всех проблем, всегда найдутся отличия операционных систем, которые нужно будет обрабатывать особым образом.

Наконец, вы можете принять различия между системами как данность и обходить их с помощью аккуратного выбора функций и макросов. Я приведу пример такого подхода в этой главе.

Таким образом, рассудительно используя переменные и функции, определяемые пользователем, минимизируя использование экзотических возможностей и полагаясь на стандартные инструменты, мы можем увеличить переносимость наших *makefile*'ов. Как уже было замечено, идеальная переносимость недостижима, поэтому нашей задачей является нахождение баланса между затратами и переносимостью. Однако прежде, чем мы исследуем специфические техники, давайте произведём обзор основных проблем переносимости *makefile*'ов.

7.1 Проблемы переносимости

Проблемы переносимости может быть нелегко охарактеризовать, поскольку они могут варьироваться от тотальной смены парадигмы (например, отличие классической Mac OS от System V Unix) до исправлений тривиальных ошибок (таких, как исправление кода возврата программы). Тем не менее, ниже перечислены основные проблемы переносимости, с которыми рано или поздно сталкивается любой *makefile*:

Имена программ
: Довольно часто в различных платформах для программ, реализующих схожую (или даже одинаковую) функциональность, используются различные имена. Наиболее ярким примером являются имена компиляторов языков C и C++ (например, *cc* и *xlc*). Также общим является добавления префикса *g* для GNU-версий программ, установленных на не GNU системах (например, *gmake*, *gawk*).

Пути
: Расположение файлов и программ варьируется от платформы к платформе. Например, в операционной системе Solaris каталогом X-сервера является */usr/X*, в то время как на многих других системах этим каталогом является */usr/X11R6*. К тому же, различие между */bin*, */usr/bin*, */sbin* и */usr/sbin* часто неясно при переходе от одной системе к другой.

Аргументы командной строки
: Аргументы командной строки программы могут отличаться, в частности при

использовании альтернативной реализации. Более того, если на какой-то платформе отсутствует нужная вам программа (или присутствующая версия этой программы вам не подходит), вам, возможно, придётся заменить эту программу другой, использующей другие аргументы командной строки.

Возможности интерпретатора

По умолчанию *make* выполняет командные сценарии с помощью */bin/sh*, однако возможности различных реализаций интерпретатора *sh* варьируются от платформы к платформе. В частности, интерпретаторы, выпущенные до принятия стандарта POSIX, не имеют множества возможностей и не принимают синтаксис современных интерпретаторов.

У Open Group есть очень полезная статья, описывающая различия между интерпретаторами System V и POSIX. Её можно найти по адресу *http://www.unix-systems.org/whitepapers/shdiffs.html*. Те, кому интересны детали, смогут найти спецификацию командного интерпретатора POSIX по адресу *http://www.opengroup.org/onlinepubs/007904975/utilities/xcu_chap02.html*

Поведение программ

Переносимым *makefile*'ам приходится бороться с программам, которые ведут себя по-разному на различных платформах. Это встречается повсеместно, поскольку различные поставщики исправляют (и совершают) ошибки и добавляют новые возможности. Существуют также обновления программ, которые поставщик может включить или не включить в релиз. Например, в 1987 году программа *awk* сменила старший номер версии. Тем не менее, даже спустя двадцать лет некоторые системы всё ещё не используют новую версию в качестве стандартной программы *awk*.

Операционная система

Наконец, существуют проблемы переносимости, связанные с совершенно различными операционными системами, например, Windows и Unix, Linux и VMS.

7.2 Cygwin

Несмотря на то, что есть порт *make* под Win32, это лишь малая часть проблемы переноса *makefile*'ов на Windows, поскольку командным интерпретатором, используемым этим портом по умолчанию, является *cmd.exe* (или *command.exe*). Это, наряду с отсутствием большинства инструментов Unix, делает реализацию кросс-платформенной переносимости очень сложной задачей. К счастью, проект

Cygwin (*http://www.cygwin.com*) реализовал для Windows библиотеку совместимости с Linux, с использованием которой были перенесены многие программы. Я уверен, что Windows разработчики, желающие достичь совместимости с Linux или получить доступ к инструментам GNU, не найти смогут лучшего инструмента.

Я использовал Cygwin на протяжении десяти лет для различных проектов, начиная с CAD-приложения, построенного на смеси C++ и Lisp, и заканчивая приложением для управления рабочим процессом, написанным на чистом Java. Набор инструментов Cygwin включает компиляторы и интерпретаторы многих языков программирования. Однако Cygwin можно выгодно использовать даже в том случае, если приложение реализовано без использования набора компиляторов и интерпретаторов Cygwin. Набор инструментов Cygwin можно использовать как вспомогательное средство для координации процессов разработки и сборки. Другими словами, совсем не обязательно писать «Cygwin» приложение или использовать средства разработки Cygwin, чтобы извлечь выгоду из Cygwin-окружения.

Тем не менее, Linux — это не Windows (слава богам!), поэтому при использовании Cygwin инструментов применительно к «родным» приложениям Windows возникает ряд проблем. Практически все эти проблемы решаются на уровне символов окончаний строки в файлах и форм путей к файлам, передающихся между Cygwin и Windows.

Окончания строк

Файловая система Windows использует для индикации окончания строки последовательность из двух символов: символа возврата каретки и символа окончания строки (CRLF). POSIX системы используют для этой цели один символ — символ окончания строки (LF). Иногда это различие может стать причиной удивления, если программа вдруг сообщит о синтаксической ошибке или перейдёт к неверной позиции в файле. Библиотека Cygwin делает всё возможное для избежания этих неприятностей. Во время установки Cygwin (или при использовании команды *mount*) вы можете выбрать, следует ли Cygwin выполнять преобразование файлов, содержащих последовательность CRLF в качестве индикатора окончания строки. Если выбран формат файлов DOS, Cygwin будет заменять последовательной CRLF на символ LF при открытии файла и производить обратное преобразование при записи текста, таким образом, UNIX-программы могут правильно работать с текстовыми файлами DOS. Если вы планируете использовать родные инструменты наподобие Visual C++ или Sun Java SDK, выбирайте формат файлов DOS. Если же вы планируете использовать компиляторы Cygwin, используйте формат UNIX (вы сможете изменить своё решение в любое время).

В добавок ко всему, Cygwin поставляется с набором инструментов для перевода форматов файлов. Программы *dos2unix* и *unix2dos* помогут преобразовать файлы в нужный формат в случае необходимости.

Путь Windows	Путь Cygwin	Альтернативный путь Cygwin
c:\usr\cygwin	*/*	*/cygdrive/c/usr/cygwin*
c:\Program Files	*/cygdrive/c/Program Files*	
c:\usr\cygwin\bin	*/bin*	*/cygdrive/c/usr/cygwin/bin*

Таблица 7.1: Стандартное отображение каталогов Cygwin

Путь Windows	Путь Cygwin	Альтернативный путь Cygwin
c:\usr\cygwin	*/*	*/c/usr/cygwin*
c:\Program Files	*/c/Program Files*	
c:\usr\cygwin\bin	*/bin*	*/c/usr/cygwin/bin*

Таблица 7.2: Модифицированное отображение каталогов Cygwin

Файловая система

Cygwin предоставляет POSIX-взгляд на файловую систему Windows. Корневой каталог файловой системы POSIX, /, отображается в каталог, в который установлен Cygwin. Диски Windows доступны из псевдокаталога */cygdrive/**буква***. Таким образом, если Cygwin установлен в каталог *C:\usr\cygwin* (я предпочитаю именно этот каталог), будет производиться отображение каталогов, представленное в таблице 7.1.

Поначалу такое преобразование может быть немного непривычным, однако на работу программ оно никак не влияет. Cygwin также предоставляет команду *mount*, позволяющую пользователям получать доступ к файлам и каталогам более удобным способом. Опция *mount* `--change-cygdrive-prefix` позволяет вам изменить префикс. Мне кажется, что изменение префикса на / может быть полезно, поскольку в этом случае доступ к дискам становится более естественным:

```
$ mount --change-cygdrive-prefix /
$ ls /c
AUTOEXEC.BAT            IO.SYS                    WINDOWS
BOOT.INI                MSDOS.SYS                 WUTemp
CD                      NTDETECT.COM              hp
CONFIG.SYS              PERSIST                   ntldr
C_DILLA                 Program Files             pagefile.sys
Documents and Settings  RECYCLER                  tmp
Home                    System Volume Information usr
I386                    Temp                      work
```

Как только вы произведёте это действие, предыдущее отображение каталогов поменяется на отображение, представленное в таблице 7.2.

Если вам нужно передать имя файла Windows-программе (например, компилятору Visual C++), вы можете просто передать относительный путь к файлу, использовав POSIX стиль, предполагающий использование прямых слэшей. Win32

API не различает прямых и обратных слэшей. К сожалению, некоторые программы, осуществляющие разбор аргументов командной строки, интерпретируют все прямые слэши как опции. Одной из таких программ является команда DOS *print*, ещё одним примером является команда *net*.

Если же используется абсолютный путь, синтаксис, основанный на именах дисков, всегда вызывает проблемы. Несмотря на то, что программы Windows обычно легко воспринимают прямые слэши в именах файлов, они совершенно не способны воспринять синтаксис */c*. Имя диска всегда должно преобразовываться в формат *c:*. Для осуществления прямых и обратных преобразований путей POSIX в пути Windows Cygwin предоставляет команду *cygpath*:

```
$ cygpath --windows /c/work/src/lib/foo.c
c:\work\src\lib\foo.c
$ cygpath --mixed /c/work/src/lib/foo.c
c:/work/src/lib/foo.c
$ cygpath --mixed --path "/c/work/src:/c/work/include"
c:/work/src;c:/work/include
```

Опция `--windows` преобразует заданный путь POSIX в путь Windows (или, при указании соответствующего аргумента, наоборот). Я предпочитаю использовать опцию `-mixed`, возвращающую путь Windows, в котором все обратные слэши заменены на прямые (таком образом, этот путь может использоваться для работы с программами Windows). Такие пути гораздо удобнее использовать в командном интерпретаторе Cygwin, воспринимающем обратный слэш как символ экранирования. Программа *cygpath* имеет множество опций, предоставляющих переносимый доступ к важным каталогам Windows:

```
$ cygpath --desktop
/c/Documents and Settings/Owner/Desktop
$ cygpath --homeroot
/c/Documents and Settings
$ cygpath --smprograms
/c/Documents and Settings/Owner/Start Menu/Programs
$ cygpath --sysdir
/c/WINDOWS/SYSTEM32
$ cygpath --windir
/c/WINDOWS
```

Если вы используете *cygpath* в смешанной Windows/Unix среде, вы можете захотеть обернуть его вызовы в переносимые функции:

```
ifdef COMSPEC
  cygpath-mixed         = $(shell cygpath -m "$1")
  cygpath-unix          = $(shell cygpath -u "$1")
```

```
  drive-letter-to-slash = /$(subst :,,$1)
else
  cygpath-mixed         = $1
  cygpath-unix          = $1
  drive-letter-to-slash = $1
endif
```

Если вам нужно только преобразовать букву диска в POSIX форму, функция *drive-letter-to-slash* будет работать быстрее, чем запуск программы *cygpath*.

Наконец, Cygwin не может спрятать все причуды Windows. Имена файлов, недопустимые в Windows, также недопустимы в Cygwin. Например, такие имена, как *aux.h*, *com1* и *prn* не могут использоваться в POSIX путях, даже при наличии расширения.

Конфликты программ

Несколько программ Windows имеют точно такие же имена, что и Unix-программы. Разумеется, программы Windows не принимают тех же самых аргументов командной строки и не ведут себя совместимым с Unix-программами образом. Если вы случайно вызвали Windows версию программы, обычным результатом является серьёзное недоумение. Наиболее проблемными программами в этом плане являются *find*, *sort*, *ftp* и *telnet*. Для достижения максимальной переносимости убедитесь в том, что вы используете абсолютные пути к этим программам при работе с Unix, Windows и Cygwin.

Если вы тесно используете Cygwin и для сборки вам не нужны базовые инструменты Windows, вы можете спокойно поместить каталог */bin* в начало переменной окружения `PATH`. Это будет гарантией того, что в первую очередь будут использоваться инструменты Cygwin, а не их Windows аналоги.

Если ваш *makefile* использует инструменты Java, учтите, что Cygwin включает программу GNU *jar*, не совместимую по формату со стандартными Sun *jar* файлами. Поэтому каталог Java jdk *bin* следует поместить в вашей переменной `PATH` раньше каталога Cygwin */bin*. Это поможет вам избежать использования программы Cygwin *jar*.

7.3 Управление программами и файлами

Наиболее общий способ управления программами заключается в использовании переменных для имён программ или путей, которые могут измениться. Переменные могут быть определены в простом блоке, как мы уже видели прежде:

```
MV ?= mv -f
RM ?= rm -f
```

или же в условном блоке:

```
ifdef COMSPEC
  MV ?= move
  RM ?= del
else
  MV ?= mv -f
  RM ?= rm -f
endif
```

Если используется простой блок, значения переменных могут измениться при использовании аргументов командной строки, при редактировании *makefile*'а или (именно для этого случая мы использовали оператор условного присваивания `?=`) при наличии соответствующей переменной окружения. Как уже было ранее замечено, одним из способов определения текущей платформы является проверка существования переменной `COMSPEC`, используемой всеми версиями операционной системы Windows. Иногда в коррекции нуждаются только пути:

```
ifdef COMSPEC
  OUTPUT_ROOT := d:
  GCC_HOME    := c:/gnu/usr/bin
else
  OUTPUT_ROOT := $(HOME)
  GCC_HOME    := /usr/bin
endif

OUTPUT_DIR := $(OUTPUT_ROOT)/work/binaries
CC := $(GCC_HOME)/gcc
```

Этот стиль приводит к *makefile*'ам, в которых бóльшая часть программ вызывается при помощи переменных *make*. Пока вы не привыкните к этому, читать такие *makefile*'ы будет немного сложнее. Однако использовать переменные в любом случае разумнее, поскольку их имена могут быть значительно короче, чем имена программ, в частности, если используются абсолютные пути.

Та же техника может быть использована для управления опциями командной строки. Например, встроенные правила содержат переменную `TARGET_ARCH`, которая может быть использована для указания опций, зависящих от платформы:

```
ifeq "$(MACHINE)" "hpux-hppa"
  TARGET_ARCH := -mdisable-fpregs
endif
```

При определении собственных программных переменных можно использовать подобный подход:

```
MV := mv $(MV_FLAGS)

ifeq "$(MACHINE)" "solaris-sparc"
  MV_FLAGS := -f
endif
```

Если вы переносите продукт на несколько платформ, цепочки секций условной обработки могут стать неуклюжими и трудными в поддержке. Вместо использования директивы `ifdef` поместите каждый набор переменных, зависящих от платформы, в собственный файл, имя которого содержит название платформы. Например, если вы определяете платформу по параметрам команды *uname*, можете выбрать соответствующий файл для включения следующим образом:

```
MACHINE := $(shell uname -smo | sed 's/ /-/g')
include $(MACHINE)-defines.mk
```

Имена файлов, содержащие пробелы, являются особенно раздражающей проблемой при использовании *make*. Предположение о том, что пробелы используются для разделения символов при синтаксическом разборе, является для *make* фундаментальным. Множество встроенных функций, таких как *word*, *filter* и *wildcard*, предполагают, что их аргументами является список слов, разделённых пробелами. Тем не менее, есть несколько приёмов, которые могут немного помочь в этом вопросе. Первый приём, описанный в разделе Поддержка нескольких каталогов бинарных файлов главы 8, заключается в замене пробелов другими символами при помощи функции *subst*:

```
space = $(empty) $(empty)
# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)
```

Функция *space-to-question* заменяет все пробелы символом вопросительного знака, используемым командным интерпретатором при определении шаблонов. Теперь мы можем реализовать функции *wildcard* и *file-exists*, умеющие правильно работать с пробелами:

```
# $(call wildcard-spaces,file-name)
wildcard-spaces = $(wildcard $(call space-to-question,$1))

# $(call file-exists,file-name)
file-exists = $(strip                                  \
                $(if $1,,$(warning $1 has no value)) \
                $(call wildcard-spaces,$1))
```

Функция *wildcard-spaces* использует *space-to-question* для осуществления операции поиска по шаблону, содержащему пробелы. Можно использовать функцию *wildcard-spaces* для реализации функции *file-exists*. Разумеется, использование символа знака вопроса может привести к тому, что функция *wildcard-spaces* будет возвращать файлы, не соответствующие первоначальному шаблону поиска (например, «my documents.doc» и «my-documents.doc»), однако вряд ли можно найти что-то получше.

Функцию *space-to-question* можно использовать для преобразования имён файлов, содержащих пробелы, в спецификациях целей и реквизитов, поскольку они допускают использование шаблонов:

```
space := $(empty) $(empty)

# $(call space-to-question,file-name)
space-to-question = $(subst $(space),?,$1)

# $(call question-to-space,file-name)
question-to-space = $(subst ?,$(space),$1)
$(call space-to-question,foo bar): $(call space-to-question,bar baz)
       touch "$(call question-to-space,$@)"
```

Если файл «*bar baz*» существует, при первом выполнении *makefile*'а реквизит будет найден, поскольку существующий файл соответствует шаблону. Однако поиск файла по шаблону, соответствующего цели, закончится неудачей, поскольку файл цели не существует. В результате переменная `$@` примет значение `foo?bar`. После этого командный сценарий вызовет функцию *question-to-space*, чтобы преобразовать значение переменной `$@` обратно в имя файла, содержащее пробел. При следующем запуске файл цели, содержащий в имени пробел, будет найден по шаблону. Этот приём выглядит немного неуклюже, однако я нашёл ему применение в реальных *makefile*'ах.

Структура каталогов исходного кода

Другим аспектом переносимости является возможность предоставления разработчикам свободы в управлении средой разработки по собственному усмотрению. Если система сборки будет требовать от них, к примеру, помещать исходный код, бинарные файлы, библиотеки и инструменты разработки в один и тот же каталог или диск Windows, рано или поздно возникнут проблемы. В конце концов, разработчики, ограниченные в дисковом пространстве, будут вынуждены разделить эти файлы.

Вместо этого имеет смысл реализовать *makefile* с использованием переменных для хранения коллекций файлов и инициализировать эти переменные разумными значениями по умолчанию. Для доступа к каждой используемой библиотеке или инструменту может быть использована соответствующая переменная, это позволит разработчикам настраивать местоположение файлов по собственному усмотрению. Используйте оператор условного присваивания при определении таких переменных, это даст разработчикам простой способ переопределения их значений через переменные окружения.

К тому же, возможность простой поддержки нескольких копий дерева каталогов с исходными и бинарными файлами является благом для разработчиков.

Даже если им не приходится поддерживать несколько платформ или использовать различные опции компиляции, разработчикам часто приходится работать с несколькими рабочими копиями исходного кода в целях отладки или при параллельной работе в нескольких проектах. Мы уже рассмотрели два возможных пути реализации этой возможности: использование высокоуровневых переменных окружения для идентификации корневого каталога дерева исходных и бинарных файлов, либо использование каталога, в котором находится *makefile*, в совокупности с фиксированным относительным путём для определения корневого каталога дерева бинарных файлов. Любой из этих подходов предоставляет разработчикам механизм для поддержки нескольких деревьев каталогов.

7.4 Работа с непереносимыми инструментами

Как уже было замечено, одной из альтернатив написания *makefile*'ов по принципу наименьшего общего знаменателя является адаптация стандартного набора инструментов. Разумеется, цель этого подхода — убедиться в том, что стандартный набор инструментов по меньшей мере так же переносим, как и ваше приложение. Очевидным выбором переносимых инструментов является набор программ проекта GNU, однако существует довольно много проектов переносимых инструментов. Два других инструмента, приходящие на ум — Perl и Python.

При отсутствии переносимых инструментов хорошей альтернативой является инкапсуляция непереносимых инструментов в функции *make*. Например, для поддержки различных компиляторов Enterprise JavaBeans (каждый из которых имеет собственный синтаксис вызова), мы можем написать функцию для компиляции архива EJB и параметризовать её для возможности подключения другого компилятора.

```
EJB_TMP_JAR = $(TMPDIR)/temp.jar
# $(call compile-generic-bean, bean-type, jar-name,
#        bean-files-wildcard, manifest-name-opt )
define compile-generic-bean
  $(RM) $(dir $(META_INF))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml %.xmi, $3),             \
    cp $(filter %.xml %.xmi, $3) $(META_INF))
  $(call compile-$1-bean-hook,$2)
  cd $(OUTPUT_DIR) &&                         \
  $(JAR) -cf0 $(EJB_TMP_JAR)                  \
         $(call jar-file-arg,$(META_INF))     \
         $(call bean-classes,$3)
  $(call $1-compile-command,$2)
  $(call create-manifest,$(if $4,$4,$2),,)
endef
```

Первым аргументом этой общей функции компиляции EJB — это тип компилятора компонентов, такого как Weblogic, Websphere и т.д. Остальными аргументами являются имя архива, список файлов архива (включая конфигурационные файлы) и необязательный файл манифеста. Сначала шаблонная функция создаёт пустой временный каталог, удаляя и создавая заново предыдущий временный каталог. Затем функция производит копирование *xml* и *xmi* файлов, указанных в качестве реквизитов каталога `$(META_INF)`. На данном этапе нам может понадобиться осуществление вспомогательных действий, будь то очистка каталога *META-INF* или подготовка *.class* файлов. Для поддержки этих операций мы включили функцию-триггер, *compile-$1-bean-hook*, которую пользователь может реализовать по собственному усмотрению. Например, если компилятор Websphere требует дополнительный контрольный файл, например, *xsl* файл, мы можем реализовать триггер следующим образом:

```
# $(call compile-websphere-bean-hook, file-list)
define compile-websphere-bean-hook
  cp $(filter %.xsl, $1) $(META_INF)
endef
```

Просто определив эту функцию, мы убеждаемся в том, что вызов *call* в функции *compile-generic-bean* будет осуществлён успешно. Если не будем писать триггер, соответствующий вызов в *compile-generic-bean* вычислится в пустую строку.

Затем наша функция создаёт jar архив. Вспомогательная функция *jar-file-arg* производит преобразование обычного пути к файлу в конкатенацию опции `-C` и относительного пути:

```
# $(call jar-file-arg, file-name)
define jar-file-arg
  -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
endef
```

Вспомогательная функция *bean-classes* извлекает подходящий class файл из списка исходных файлов (в jar архив нужно включать только интерфейсы и home классы):

```
# $(call bean-classes, bean-files-list)
define bean-classes
  $(subst $(SOURCE_DIR)/,,                \
    $(filter %Interface.class %Home.class, \
      $(subst .java,.class,$1)))
endef
```

Затем общая функция вызывает соответствующую команду компиляции `$(call $1-compile-command,$2)`:

```
define weblogic-compile-command
  cd $(TMPDIR) && \
  $(JVM) weblogic.ejbc -compiler $(EJB_JAVAC) $(EJB_TMP_JAR) $1
endef
```

Наконец, общая функция добавляет файл манифеста.

После определения функции *compile-generic-bean* мы можем обернуть её вызов в специальную функцию для каждого компилятора, который мы хотим поддерживать.

```
# $(call compile-weblogic-bean, jar-name,
#        bean-files-wildcard, manifest-name-opt )
define compile-weblogic-bean
  $(call compile-generic-bean,weblogic,$1,$2,$3)
endef
```

Стандартный интерпретатор

Следует ещё раз подчеркнуть, что одним из самых досадных источников непереносимости при переходе на другую систему являются возможности интерпретатора */bin/sh*, используемого *make* по умолчанию. Если вам приходится настраивать командные сценарии вашего *makefile*'а, рассмотрите возможность стандартизации вашего интерпретатора. Разумеется, это не очень подходит для типичных проектов с открытым исходным кодом, *makefile*'ы которых выполняются в неконтролируемой среде. Однако в случае, если вы управляете средой и контролируете число машин, которые нужно настроить, такой подход вполне разумен.

Многие интерпретаторы предоставляют возможности, которые могут исключить использование большого числа небольших программ. Например, *bash* включает расширенные возможности работы с переменными, такие как `%%` и `##`, которые могут помочь избежать использования инструментов, таких как *sed* и *expr*.

7.5 Automake

В этой главе мы сосредоточились на использовании GNU *make* и эффективной поддержке инструментов для достижения переносимости систем сборки. Однако иногда даже эти скромные цели недостижимы. Если вы не можете использовать мощные возможности GNU *make* и вынуждены полагаться на ограниченный набор возможностей, продиктованный подходом наименьшего общего знаменателя, вам следует рассмотреть возможность использования программы *automake*, *http://www.gnu.org/software/automake/automake.html*.

Программа *automake* принимает на вход стилизованный *makefile* и производит переносимый *makefile*. Работа *automake* основывается на применении макроязыка *m4*, допускающего довольно сжатый синтаксис входных файлов (обычно их имя

makefile.am). Как правило, *automake* используется в совокупности с программой *autoconf*, пакетом поддержки переносимости программ, написанных на C/C++, однако использование *autoconf* не обязательно.

В то время как *automake* является хорошим решением для систем сборки, требующих максимальной переносимости, *makefile*'ы, которые производит эта программа, не имеют доступа к богатым возможностям GNU *make* (за исключением оператора `+=`, для поддержки которого используются особые средства). Более того, синтаксис входных файлов *automake* имеет мало общего с синтаксисом обычных *makefile*'ов. Поэтому использование *automake* (без *autoconf*) не очень сильно отличается от подхода наименьшего общего знаменателя.

Глава 8

C и C++

Проблемы и техники, показанные в главе 6, раскрываются и применяются в этой главе к проектам, написанным на C и C++. Мы продолжим рассматривать наш пример mp3 плеера, сборка которого осуществляется нерекурсивным *makefile*'ом.

8.1 Разделение исходных и бинарных файлов

Итак, что же нам делать, если мы хотим поддерживать единственное дерево каталогов с исходным кодом, но множество платформ и множество сборок для каждой платформы, разделяя при необходимости деревья каталогов исходных и бинарных файлов? Изначально программа *make* была написана для эффективной работы с файлами, находящимися в одном каталоге. И хотя со времени своего создания она очень изменилась, истоки забыты не были. Лучше всего *make* работает с несколькими каталогами, если модифицируемые им файлы находятся в текущем каталоге или в его подкаталогах.

Простой способ

Наиболее простой способ заставить *make* помещать бинарные файлы в отдельный каталог — это запустить *make* из этого каталога. Доступ к выходным файлам осуществляется через относительные пути, как показано в предыдущей главе, в то время как исходные файлы должны быть указаны явно или при помощи `vpath`. В любом случае, нам придётся ссылаться на каталог с исходными файлами из нескольких мест, поэтому нам нужно завести переменную для хранения пути к нему:

```
SOURCE_DIR := ../mp3_player
```

Возьмём за основу наш предыдущий *makefile*. Функция *source-to-object* остаётся неизменной, а вот функцию *subdirectory* нужно изменить так, чтобы она учитывала относительные пути к исходным файлам.

```
# $(call source-to-object, source-file-list)
source-to-object = $(subst .c,.o,$(filter %.c,$1)) \
                   $(subst .y,.o,$(filter %.y,$1)) \
                   $(subst .l,.o,$(filter %.l,$1))
# $(subdirectory)
subdirectory = $(patsubst $(SOURCE_DIR)/%/module.mk,%, \
                 $(word                                \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))
```

Файлы, перечисленные в переменной `MAKEFILE_LIST`, будут включать относительные пути к исходным файлам. Таким образом, чтобы получить относительный путь к каталогу модуля, нам нужно помимо суффикса *module.mk* отсечь и префикс.

Далее, чтобы помочь *make* найти исходные файлы, мы используем директиву `vpath`:

```
vpath %.y $(SOURCE_DIR)
vpath %.l $(SOURCE_DIR)
vpath %.c $(SOURCE_DIR)
```

Это позволит нам использовать для исходных файлов такие же простые относительные пути, как и для выходных файлов. Когда *make* будет искать исходный файл и не сможет найти его в текущем каталоге, он будет сканировать каталог `SOURCE_DIR`. Далее, нам нужно обновить значение переменной `include_dirs`:

```
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
```

В дополнение к каталогам исходных файлов это переменная включает подкаталог *lib* дерева бинарных файлов, ведь именно туда будут попадать заголовочные файлы, составленные *lex* и *yacc*.

Также нужно обновить директиву `include` для получения доступа к файлам *module.mk*, поскольку *make* не использует директиву `vpath` для поиска включаемых файлов:

```
include $(patsubst %,$(SOURCE_DIR)/%/module.mk,$(modules))
```

Наконец, мы создаём каталоги для размещения выходных файлов:

```
create-output-directories :=                    \
    $(shell for f in $(modules);                \
            do                                  \
              $(TEST) -d $$f || $(MKDIR) $$f;   \
            done)
```

Это присваивание создаёт пустую переменную, значение которой никогда не используется. Это даёт гарантию, что необходимые каталоги будут созданы до начала основной работы *make*. Нам приходится самим создавать каталоги, так как *lex*, *yacc* и инструменты автоматической генерации зависимостей не сделают этого за нас.

Ещё один способ убедиться в том, что необходимые каталоги созданы — добавить эти каталоги в качестве реквизитов файлов зависимостей (эти файлы имеют расширение *.d*). Однако это плохая идея, так как каталог на самом деле не является реквизитом. Файлы *yacc*, *lex* или файлы зависимостей не зависят от *содержимого* каталога, и не должны создаваться заново только от того, что временна́я метка каталога изменилась. На самом деле это будет источником неэффективности в случае добавления или удаления файлов из каталога, содержащего выходные файлы.

Изменения, которые нужно сделать в файле *module.mk*, ещё проще:

```
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library, $(subdirectory)/libdb.a, $(local_src)))

.SECONDARY: $(call generated-source, $(local_src))

$(subdirectory)/scanner.d: $(subdirectory)/playlist.d
```

Эта версия не использует функцию *wildcard* для определения исходных файлов. Пусть восстановление этого функционала станет упражнением для читателя. В первоначальном варианте был небольшой сбой. Когда я запустил этот *makefile* я обнаружил, что файл зависимостей *scanner.d* был создан до файла *playlist.h*, от которого он зависит. Эта зависимость не была отражена в исходном *makefile*'е, однако по счастливой случайности всё работало правильно. Правильное определение *всех* зависимостей является трудной задачей даже для небольших проектов.

Если предположить, что исходные файлы находятся в подкаталоге *mp3_ player*, то сборка проекта будет осуществляться следующим образом:

```
$ mkdir mp3_player_out
$ cd mp3_player_out
$ make --file=../mp3_player/makefile
```

Наш *makefile* правилен и хорошо работает, однако немного раздражает то, что приходится изменять каталог и добавлять опцию `--file` (`-f`). Эту проблему можно решить с помощью простого сценария:

```
#! /bin/bash
if [[ ! -d $OUTPUT_DIR ]]
then
```

```
  if ! mkdir -p $OUTPUT_DIR
  then
    echo "Cannot create output directory" > /dev/stderr
  exit 1
  fi
fi

cd $OUTPUT_DIR
make --file=$SOURCE_DIR/makefile "$@"
```

Этот сценарий подразумевает, что каталог с исходными файлами и каталог выходных файлов хранятся в переменных окружения `SOURCE_DIR` и `OUTPUT_DIR` соответственно. Это является обычной практикой, позволяющей разработчикам легко переключать деревья каталогов, не печатая при этом пути слишком часто.

Наш *makefile* не содержит никаких средств, позволяющих запретить разработчикам выполнять *makefile* из каталога с исходными файлами. Это частая ошибка, и некоторые командные сценарии могут вести себя неправильно. Например, цель `clean`:

```
.PHONY: clean
clean:
    $(RM) -r *
```

удалит всё дерево каталогов с исходным кодом пользователя! Благоразумным решением является добавление соответствующей проверки в самом начале выполнения *makefile*'а. Ниже приведён возможный вариант такой проверки:

```
$(if $(filter $(notdir $(SOURCE_DIR)),$(notdir $(CURDIR))),\
  $(error Пожалуйста, запустите makefile из бинарного дерева каталогов.))
```

Этот код проверяет совпадение имени текущего рабочего каталога (`$(notdir $(CURDIR))`) с именем каталога, содержащего исходный код: (`$(notdir $(SOURCE_DIR))`). Если эти два выражения совпадают, выводится сообщение об ошибке, и выполнение *make* прекращается. Поскольку результатом вычисления функций *if* и *error* является пустая строка, мы можем поместить эти две строчки кода сразу после определения переменной `SOURCE_DIR`.

Сложный способ

Некоторые разработчики находят необходимость перемещения в дерево каталогов, содержащих бинарные файлы, настолько раздражающей, что готовы проделать огромную работу для того, чтобы этого избежать. Или, возможно, разработчик *makefile*'а использует среду, в которой использование сценариев-обёрток или псевдонимов программ недопустимо. В любом случае, можно изменить *makefile* таким образом, чтобы была возможность производить запуск *make* из дерева

каталогов, содержащих исходный код, и помещать бинарные файлы в другое дерево каталогов с помощью добавления к путям выходных файлов соответствующих префиксов. Обычно в этом случае я использую абсолютные пути, поскольку такой подход предоставляет больше гибкости, хотя и обостряет проблему конечности длины командной строки. Для исходных файлов используются простые относительные пути (они вычисляются относительно каталога, в котором располагается *makefile*).

Следующий пример содержит модифицированный *makefile*, позволяющий запускать *make* из каталога с исходным кодом и записывающий бинарные файлы в отдельное дерево каталогов:

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,  \
                     $(subst .c,.o,$(filter %.c,$1)) \
                     $(subst .y,.o,$(filter %.y,$1)) \
                     $(subst .l,.o,$(filter %.l,$1)))
# $(subdirectory)
subdirectory = $(patsubst %/module.mk,%,  \
                 $(word                   \
                   $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST)))

# $(call make-library, library-name, source-file-list)
define make-library
  libraries += $(BINARY_DIR)/$1
  sources   += $2
  $(BINARY_DIR)/$1: $(call source-dir-to-binary-dir,  \
                      $(subst .c,.o,$(filter %.c,$2)) \
                      $(subst .y,.o,$(filter %.y,$2)) \
                      $(subst .l,.o,$(filter %.l,$2)))
      $(AR) $(ARFLAGS) $$@ $$^
endef

# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,   \
                     $(subst .y,.c,$(filter %.y,$1))  \
                     $(subst .y,.h,$(filter %.y,$1))  \
                     $(subst .l,.c,$(filter %.l,$1))) \
                   $(filter %.c,$1)

# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
```

```
  $(call one-compile-rule,$(call source-to-object,$f),$f))
endef

# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
      $(COMPILE.c) -o $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
      $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
      $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
      $(MV) $$@.tmp $$@
endef

modules      := lib/codec lib/db lib/ui app/player
programs     :=
libraries    :=
sources      :=

objects      = $(call source-to-object,$(sources))
dependencies = $(subst .o,.d,$(objects))

include_dirs := $(BINARY_DIR)/lib lib include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))
vpath %.h $(include_dirs)

MKDIR := mkdir -p
MV    := mv -f
RM    := rm -f
SED   := sed
TEST  := test

create-output-directories :=                                        \
    $(shell for f in $(call source-dir-to-binary-dir,$(modules)); \
            do                                                      \
              $(TEST) -d $$f || $(MKDIR) $$f;                       \
            done)

all:

include $(addsuffix /module.mk,$(modules))

.PHONY: all
all: $(programs)

.PHONY: libraries
libraries: $(libraries)

.PHONY: clean
clean:
```

```
    $(RM) -r $(BINARY_DIR)

ifneq "$(MAKECMDGOALS)" "clean"
include $(dependencies)
endif
```

В этой версии функция *source-to-object* модифицирована так, чтобы осуществлять исправление путей к бинарным файлам. Поскольку эта операция осуществляется несколько раз, реализуем её в виде функции:

```
SOURCE_DIR := /test/book/examples/ch07-separate-binaries-1
BINARY_DIR := /test/book/out/mp3_player_out

# $(call source-dir-to-binary-dir, directory-list)
source-dir-to-binary-dir = $(addprefix $(BINARY_DIR)/, $1)

# $(call source-to-object, source-file-list)
source-to-object = $(call source-dir-to-binary-dir,  \
                     $(subst .c,.o,$(filter %.c,$1)) \
                     $(subst .y,.o,$(filter %.y,$1)) \
                     $(subst .l,.o,$(filter %.l,$1)))
```

Функция *make-library* изменена подобным образом, теперь она добавляет к выходным файлам префикс `BINARY_DIR`. Теперь мы используем предыдущую функции *subdirectory*, поскольку путь к каталогам с включаемыми файлами снова стал простым относительным путём. Есть одна небольшая загвоздка: ошибка в *make* 3.80 препятствует вызову функции *source-to-object* из новой версии функции *make-library*. Эта ошибка была исправлена в версии 3.81. Мы можем обойти эту ошибку, подставив вместо вызова функции *source-to-object* её тело.

Рассмотрим теперь по-настоящему неуклюжую часть. Неявные правила не работают, если доступ к выходному файлу нельзя осуществить с помощью относительного пути. Например, основное правило компиляции `%.o: %.c` отлично работает, когда оба файла находятся в одном каталоге, или даже если исходный файл располагается в каком-нибудь подкаталоге, например, *lib/codec/codec.c*. Если исходный файл располагается в удалённом каталоге, мы можем указать *make* на необходимость его поиска при помощи директивы `vpath`. Однако если объектный файл располагается в удалённом каталоге, *make* не сможет определить местоположения этого файла, и цепочка правил будет нарушена. Единственный способ информировать *make* о расположении выходного файла — это предоставить явное правило, соединяющее исходный и объектный файлы:

```
$(BINARY_DIR)/lib/codec/codec.o: lib/codec/codec.c
```

Такая операция должна быть осуществлена для каждого объектного файла.

Хуже того, эта пара не соответствует неявному правилу `%.o: %.c`. Это значит, что нам самим нужно предоставить командный сценарий, повторяющий правило встроенной базы правил, и, возможно, повторить этот сценарий много раз. Проблема также касается правила автоматического составления файлов зависимостей, которое мы используем. Добавление двух явных правил для каждого объектного файла, упоминающегося в *makefile*'е, — это кошмар для человека, который будет поддерживать программный продукт. Однако мы можем минимизировать дублирование кода, написав функцию для автоматического составления этих правил:

```
# $(call one-compile-rule, binary-file, source-files)
define one-compile-rule
  $1: $(call generated-source,$2)
      $(COMPILE.c) $$@ $$<

  $(subst .o,.d,$1): $(call generated-source,$2)
      $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $$< | \
      $(SED) 's,\($$(notdir $$*)\.o\) *:,$$(dir $$@)\1 $$@: ,' > $$@.tmp
      $(MV) $$@.tmp $$@
endef
```

Первые две строки функции — это явные правила для описания зависимости объектного файла от исходного. Реквизит для этого правила должен быть вычислен при помощи функции *generated-source*, описанной в главе 6. Если этого не сделать, исходные файлы *lex* и *yacc* вызовут ошибку компиляции при появлении в командном сценарии (после подстановки переменной `$^`). Автоматические переменные экранированы, поэтому они будут вычислены позднее, при выполнении командного сценария (а не при выполнении функции *eval*). Функция *generated-source* была модифицирована так, чтобы возвращать пути к исходным файлам, написанным на языке C, неизменными:

```
# $(call generated-source, source-file-list)
generated-source = $(call source-dir-to-binary-dir,   \
                     $(subst .y,.c,$(filter %.y,$1))  \
                     $(subst .y,.h,$(filter %.y,$1))  \
                     $(subst .l,.c,$(filter %.l,$1))) \
                   $(filter %.c,$1)
```

С учётом этого изменения функция будет возвращать результаты, представленные в следующей таблице:

Аргумент	Результат
lib/db/playlist.y	*/c/mp3_player_out/lib/db/playlist.c*
	/c/mp3_player_out/lib/db/playlist.h
lib/db/scanner.l	*/c/mp3_player_out/lib/db/scanner.c*
app/player/play_mp3.c	*app/player/play_mp3.c*

Явное правило для создания файлов зависимостей устроено точно так же. Ещё раз обратите внимание на экранирование (двойные символы доллара), которое требуется для правильной работы сценария.

Теперь для каждого исходного файла в модуле нужно вычислить нашу новую функцию:

```
# $(compile-rules)
define compile-rules
  $(foreach f, $(local_src),\
    $(call one-compile-rule,$(call source-to-object,$f),$f))
endef
```

Эта функция использует глобальную переменную `local_src`, переопределяемую в файлах *module.mk*. Более общий подход заключается в передаче списка файлов в качестве аргумента, однако в нашем проекте это делать не обязательно. Просто добавим эти функции в файлы *module.mk*:

```
local_src := $(subdirectory)/codec.c

$(eval $(call make-library,$(subdirectory)/libcodec.a,$(local_src)))

$(eval $(compile-rules))
```

Мы должны использовать функцию *eval*, так как результат вычисления функции *compile-rules* содержит более одной строки кода *make*.

Наконец, последняя сложность. Так как стандартные шаблонные правила компиляции исходных файлов C не могут определить путь к выходным файлам, неявное правило для *lex* и наше шаблонное правило для *yacc* также не смогут этого сделать. Мы можем легко изменить эти правила самостоятельно. Поскольку они больше не применимы к остальным *lex* и *yacc* файлам, мы можем вынести их в файл *lib/db/module.mk*:

```
local_dir := $(BINARY_DIR)/$(subdirectory)
local_src := $(addprefix $(subdirectory)/,playlist.y scanner.l)

$(eval $(call make-library,$(subdirectory)/libdb.a,$(local_src)))

$(eval $(compile-rules))

.SECONDARY: $(call generated-source, $(local_src))

$(local_dir)/scanner.d: $(local_dir)/playlist.d

$(local_dir)/%.c $(local_dir)/%.h: $(subdirectory)/%.y
    $(YACC.y) --defines $<
    $(MV) y.tab.c $(dir $@)$*.c
```

```
    $(MV) y.tab.h $(dir $@)$*.h

$(local_dir)/scanner.c: $(subdirectory)/scanner.l
    @$(RM) $@
    $(LEX.l) $< > $@
```

Правила для *lex* файлов реализованы как обычные явные правила, а правила для *yacc* файлов — как шаблонные. Почему? Потому что правила для *yacc* файлов используются для сборки двух целей: исходного и заголовочного файла на языке C. Если мы используем обычное явное правило, *make* выполнит командный сценарий дважды: один раз для исходного файла, а второй — для заголовочного. Однако *make* считает, что шаблонное правило, определяющее несколько целей, при выполнении обновляет обе цели. Если возможно, вместо *makefile*'ов, приведённых в этом разделе, я буду использовать более простой подход, основанный на компиляции из дерева каталогов бинарных файлов. Как вы могли заметить, при попытке компиляции из дерева исходных файлов немедленно возникают трудности, и с ростом размеров проекта эти трудности только растут.

8.2 Объявляем права «только для чтения»

Как только деревья каталогов с исходными и бинарными файлами разделены, возможность объявления прав доступа «только для чтения» для справочного дерева каталогов с исходным кодом получается практически бесплатно, если только все объектные файлы, создаваемые сборкой, помещаются в дерево каталогов бинарных файлов. Однако если при сборке создаются исходные файлы, мы должны позаботиться о том, чтобы они также были помещены в дерево каталогов бинарных файлов.

В более простом подходе, основанном на компиляции в бинарном дереве, все создаваемые файлы помещались в бинарное дерево автоматически, так как именно из него происходил вызов программ *lex* и *yacc*. При подходе, основанном на компиляции из дерева каталогов с исходными файлами, мы были вынуждены указывать явные пути для исходных и целевых файлов, поэтому спецификация пути к файлу в бинарном дереве каталогов не потребует дополнительной работы, нужно просто не забыть сделать это.

Источником остальных препятствий для объявления дерева каталогов с исходными файлами доступным только для чтения обычно является среда. Часто система сборки, доставшаяся вам по наследству, содержит действия, создающие файлы в дереве каталогов с исходными файлами, так как первоначальный её автор не осознал преимущества исходного дерева, доступного только для чтения. В качестве примеров можно привести автоматически составленную документацию, файлы журналов и временные файлы. Перемещение этих файлов в дерево каталогов с бинарными файлами может потребовать значительных усилий, однако, если

нужна поддержка нескольких деревьев каталогов бинарных файлов, эта жертва является необходимой. Альтернативой является поддержка и синхронизация нескольких идентичных деревьев каталогов с исходными файлами.

8.3 Генерация зависимостей

Небольшое введение в технику автоматической генерации зависимостей можно найти в разделе «Автоматическое определение зависимостей» главы 2, однако это введение умалчивает о ряде важных проблем. Этот раздел описывает несколько альтернатив простого решения, рассмотренного нами ранее[1]. В частности, описанный ранее подход, предлагаемый руководством пользователя GNU *make*, обладает следующими недостатками:

- Он неэффективен. Когда *make* обнаруживает, что файл зависимостей не существует или устарел, он обновляет этот файл и стартует заново. Повторное чтение *makefile*'а может быть неэффективным, если во время чтения осуществляется много задач или требуется анализ графа зависимостей.

- Каждый раз при запуске сборки после добавления новых исходных файлов *make* выдаёт предупреждение. На момент запуска файл зависимостей, ассоциированный с новым исходным файлом, ещё не существует, поэтому при попытке прочитать этот файл зависимостей *make* выдаст предупреждение до того, как осуществит генерацию файла. Это не критично, но порой очень раздражает.

- Если вы удалите исходный файл, при попытке осуществления сборки *make* будет завершать своё выполнение с ошибкой. Причиной ошибки является существование файл зависимостей, содержащего удалённый исходный файл в качестве реквизита. Поскольку *make* не может найти удалённый файл и не имеет правила для его сборки, выдаётся следующее сообщение об ошибке:

  ```
  make: *** No rule to make target foo.h, needed by foo.d. Stop.
  ```

  Более того, из-за этой ошибки *make* не сможет обновить файл зависимостей. Единственный возможный выход — удалить файл зависимостей вручную, однако обычно найти эти файлы нелегко, и пользователи, как правило, удаляют все файлы зависимостей и осуществляют чистую сборку. Та же проблема возникает при переименовании файлов.

[1] Бóльшая часть материала, рассмотренного в этом разделе, была разработана Томом Троми (Tom Tromey, tromey@cygnus.com) для проекта GNU *automake* и взята из прекрасной сводной статьи Пола Смита (Paul Smith, программист, осуществляющий поддержку GNU *make*), прочитать которую можно на его веб-сайте ***http://make.paulandlesley.org***. (прим. автора)

Обратите внимание на то, что эта проблема чаще всего появляется при удалении или переименовании заголовочных (*.h*) файлов. Причина этого заключается в том, что *.c* файлы будут удалены из списка зависимостей автоматически и не вызовут проблем при сборке.

8.3.1 Решение Троми

Давайте разбирать проблемы по очереди.

Как нам избежать повторного запуска *make*?

После небольшого размышления можно понять, что повторный запуск *make* не требуется. Если файл зависимостей обновлён, это значит, что хотя бы один его реквизит изменился, что, в свою очередь, значит, что нам нужно собрать целевой файл заново. На данном этапе *make* не нуждается в дополнительной информации о зависимостях, поскольку она не изменит его поведения. Однако нам нужно, чтобы файл зависимостей был обновлён, чтобы при следующем запуске *make* обладал полной информацией о зависимостях.

Поскольку в текущей сборке нам не нужен файл зависимостей, мы можем обновить его при сборке целевого файла. Мы можем добиться этого путём соответствующего изменения правила компиляции:

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 | \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(MV) $3.tmp $3
endef

%.o: %.c
    $(call make-depend,$<,$@,$(subst .o,.d,$@))
    $(COMPILE.c) -o $@ $<
```

Мы реализовали возможность создания файлов зависимостей в форме функции *make-depend*, принимающей в качестве аргументов имена исходного и объектного файлов, а также имя файла зависимостей. Это предоставляет нам максимальную гибкость на случай, если мы решим повторно использовать эту функцию в другом контексте. После подобного изменения правила компиляции следует удалить шаблонное правило `%.d: %.c`, это позволит избежать повторного составления файла зависимостей.

Теперь объектный файл и файл зависимостей логически связаны: если существует один из них, должен существовать и второй. Таким образом, нам не нужно больше беспокоиться об отсутствии файла зависимостей. Если он не существует, объектный файл тоже не существует, оба этих файла будут созданы при следующей сборке. Теперь мы можем игнорировать любые предупреждения, возникающие из-за отсутствия файлов зависимостей.

В разделе «Условная обработка и включения» главы 3 была описана альтернативная форма директивы `include`, `-include` или (`sinclude`), игнорирующая ошибки и не выдающая предупреждений:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif
```

Это решает вторую проблему — раздражающие сообщения, возникающие при отсутствии файлов зависимостей.

Наконец, мы можем избежать предупреждений об отсутствующих реквизитах с помощью небольшого трюка. Трюк заключается в спецификации для отсутствующего файла цели без реквизитов и команд. Предположим для примера, что генератор зависимостей создал следующую зависимость:

```
target.o target.d: header.h
```

Допустим теперь, что в результате рефакторинга кода файл *header.h* был удалён. При следующем запуске *makefile*'а мы получим следующую ошибку:

```
make: *** No rule to make target header.h, needed by target.d. Stop.
```

Однако если мы добавим цель `header.h`, не имеющую ассоциированного командного сценария, ошибки не будет:

```
target.o target.d: header.h
header.h:
```

Это происходит потому, что если файл *header.h* не существует, он просто помечается устаревшим и все цели, имеющие этот файл в качестве реквизита, собираются заново. Таким образом, файл зависимостей будет создан заново и уже не будет содержать *header.h*. Если же файл *header.h* существует, *make* просто продолжит выполнение. Теперь всё, что нужно сделать — это убедиться в том, что каждый реквизит имеет соответствующее пустое правило. Этот вид привил впервые встретился нам в разделе «Абстрактные цели» главы 2. Ниже приведена версия функции *make-depend*, добавляющая новую цель:

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -M $1 |        \
  $(SED) 's,\($$(notdir $2)\) *:,$$(dir $2) $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                       \
         -e 's/^[^:]*: *//'                                 \
         -e 's/ *\\$$$$//'                                  \
         -e '/^$$$$/ d'                                     \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

Чтобы создать дополнительные правила, мы применяем новую команду *sed* к файлу зависимостей. Этот кусок кода *sed* осуществляет пять преобразований:

1. Удаляет комментарии.
2. Удаляет целевые файлы и соответствующие пробелы.
3. Удаляет оконечные пробелы.
4. Удаляет пустые строки.
5. Добавляет в конец каждой строки двоеточие.

(GNU *sed* может читать файл и добавлять к нему текст в одной команде, предохраняя нас от необходимости использования второго временного файла. Этот код может не работать в других системах.) Новая версия команды *sed* принимает на вход текст следующего вида:

```
# любые комментарии
target.o target.d: prereq1 prereq2 prereq3 \
    prereq4
```

и преобразует его к виду:

```
prereq1 prereq2 prereq3:
prereq4:
```

Таким образом, функция *make-depend* добавляет новые цели к исходному файлу зависимостей. Это решает проблему «No rule to make target».

8.3.2 Программы *makedepend*

Всё это время мы могли использовать опцию `-M`, которой обладает бóльшая часть компиляторов, но что бы мы делали, если бы этой опции не существовало? Кроме того, есть ли более удачные решения, чем использование опции `-M`?

На данный момент практически все компиляторы языка C имеют поддержку генерации зависимостей исходных файлов, однако не так давно всё было иначе. На заре проекта X Window System его разработчики создали программу *makedepend*, определяющую зависимости для заданного набора исходных файлов C и C++. Доступ к этой программе можно получить бесплатно через сеть Internet. Использовать эту программу немного неудобно, поскольку она написана так, чтобы добавлять свой вывод в *makefile*, чего нам не хотелось бы. Программа *makedepend* подразумевает, что объектные файлы располагаются в том же каталоге, что и исходные. Это, в свою очередь, означает, что нам нужно изменить сценарий *sed*:

```
# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(MAKEDEPEND) -f- $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) $1 | \
  $(SED) 's,^.*/\([^/]*\.o\) *:,$(dir $2)\1 $3: ,' > $3.tmp
  $(SED) -e 's/#.*//'                                          \
         -e 's/^[^:]*: *//'                                    \
         -e 's/ *\\$$$$//'                                     \
         -e '/^$$$$/ d'                                        \
         -e 's/$$$$/ :/' $3.tmp >> $3.tmp
  $(MV) $3.tmp $3
endef
```

Опция `-f-` программы *makedepend* означает, что информация о зависимостях должна выводиться на стандартный поток вывода.

Альтернативой использованию *makedepend* или вашего собственного компилятора является использование компилятора *gcc*. Этот компилятор имеет огромное количество опций для составления информации о зависимостях. Наиболее подходящими для нашего случая выглядят опции, используемые в следующем примере:

```
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(dependencies)
endif

# $(call make-depend,source-file,object-file,depend-file)
define make-depend
  $(GCC) -MM            \
         -MF $3         \
         -MP            \
         -MT $2         \
         $(CFLAGS)      \
         $(CPPFLAGS)    \
         $(TARGET_ARCH) \
         $1
endef

%.o: %.c
    $(call make-depend,$<,$@,$(subst .o,.d,$@))
    $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Опция `-MM` призывает *gcc* убрать все системные заголовочные файлы из списка реквизитов. Это удобно, так как эти файлы меняются редко (если вообще меняются), и помогает сократить беспорядок, возникающий с ростом и усложнением системы сборки. Изначально эта опция могла быть введена из соображений производительности. Однако при использовании современных процессоров разница в производительности едва ли может быть измерена.

Опция `-MF` специфицирует имя файла зависимостей. В качестве имени будет использоваться имя объектного файла, расширение которого заменяется на *.d*. *gcc*

имеет ещё одну опцию, `-MD` или `-MMD`, которая автоматически определяет имя файла зависимостей, используя правило, подобное описанному выше. В общем случае мы предпочли бы использовать эту опцию, однако встроенное правило компилятора не сможет добавить соответствующий относительный путь к каталогу с объектными файлами, вместо этого оно просто поместит файлы зависимостей в текущий каталог. Поэтому мы вынуждены делать эту работу самостоятельно и использовать опцию `-MF`.

Опция `-MP` призывает *gcc* включать для каждого реквизита абстрактную цель. Это делает ненужным наше неуклюжее пятизвенное выражение для *sed*, использовавшееся в функции *make-depend*. Похоже, эту опцию добавили в *gcc* по просьбе разработчиков *automake*, которые изобрели технику абстрактных целей.

Наконец, опция `-MT` специфицирует строку, которая будет использоваться для целей в файле зависимостей. Повторим, без этой опции *gcc* не сможет включить относительный путь к каталогу объектных файлов.

Используя *gcc*, мы можем заменить четыре команды, требующиеся для генерации зависимостей, одной. Даже если вы используете коммерческий компилятор, вы можете использовать *gcc* для управления зависимостями.

8.4 Поддержка нескольких каталогов бинарных файлов

После реализации *makefile*'а, осуществляющего запись бинарных файлов в отдельное дерево каталогов, реализовать поддержку множества таких деревьев довольно просто. Для интерактивных сборок, инициируемых разработчиками при помощи клавиатуры, требуется совсем мало подготовки. Разработчик создаёт каталог для бинарных файлов, переходит в него и вызывает *make*, указав нужный *makefile*.

```
$ mkdir -p ~/work/mp3_player_out
$ cd ~/work/mp3_player_out
$ make -f ~/work/mp3_player/makefile
```

Если процесс запуска сборки требует от разработчика больше участия, то сценарий-обёртка будет наилучшим решением. Этот сценарий может также анализировать текущий каталог и выставлять соответствующим образом переменные окружения, используемые в *makefile*'е (например, `BINARY_DIR`).

```
#! /bin/bash
# Работаем в каталоге с исходными файлами.
curr=$PWD
export SOURCE_DIR=$curr
while [[ $SOURCE_DIR ]]
do
  if [[ -e $SOURCE_DIR/[Mm]akefile ]]
```

```
  then
    break;
  fi
  SOURCE_DIR=${SOURCE_DIR%/*}
done

# Если makefile не найден, выводим сообщение об ошибке.
if [[ ! $SOURCE_DIR ]]
then
  printf "run-make: Cannot find a makefile" > /dev/stderr
  exit 1
fi

# Если каталог для выходных файлов не задан, используем значение
# по умолчанию.
if [[ ! $BINARY_DIR ]]
then
  BINARY_DIR=${SOURCE_DIR}_out
fi

# Создаём каталог для бинарных файлов.
mkdir --parents $BINARY_DIR

# Запускаем make.
make --directory="$BINARY_DIR" "$@"
```

Этот сценарий не очень сложен. Он производит поиск *makefile*'а в текущем каталоге, и в случае неудачи поднимается вверх по дереву каталогов, пока не найдёт *makefile*. Затем происходит проверка наличия переменной окружения, содержащей каталог для бинарных файлов. Если переменная не определена, ей присваивается значение по умолчанию, получаемое добавлением суффикса «_out» к имени каталога с исходными файлами. Затем сценарий создаёт каталог для бинарных файлов и осуществляет запуск *make*.

Если осуществляются сборки для различных платформ, требуются методы для определения нужной платформы. Наиболее простой подход требует от разработчика определения переменной окружения для каждого типа платформы и добавления условных директив, использующих эту переменную, в *makefile* и в исходный код. Лучшим подходом является автоматическое определение платформы на основании вывода программы *uname*.

```
space := $(empty) $(empty)
export MACHINE := $(subst $(space),-,$(shell uname -smo))
```

Я считаю, что при автоматическом запуске сборки программой *cron* использование вспомогательного сценария командного интерпретатора является более предпочтительным подходом, нежели прямой вызов *make*. Сценарий-обёртка предоставляет больше возможностей для подготовки, обработки ошибок и завершения

автоматизированной сборки. Сценарий также является подходящим местом для определения переменных и опций командной строки.

Наконец, если проект поддерживает фиксированное число деревьев каталогов и платформ, вы можете использовать имена каталогов для автоматического определения параметров текущей сборки. Например:

```
ALL_TREES := /builds/hp-386-windows-optimized \
             /builds/hp-386-windows-debug     \
             /builds/sgi-irix-optimzed        \
             /builds/sgi-irix-debug           \
             /builds/sun-solaris8-profiled    \
             /builds/sun-solaris8-debug

BINARY_DIR := $(foreach t,$(ALL_TREES),\
                $(filter $(ALL_TREES)/%,$(CURDIR)))

BUILD_TYPE := $(notdir $(subst -,/,$(BINARY_DIR)))

MACHINE_TYPE := $(strip              \
                  $(subst /,-,       \
                    $(patsubst %/,%, \
                      $(dir          \
                        $(subst -,/, \
                          $(notdir $(BINARY_DIR)))))))
```

Переменная `ALL_TREES` содержит список всех возможных каталогов бинарных файлов. Цикл `foreach` осуществляет проверку соответствия текущего каталога одному из возможных каталогов бинарных файлов, причём соответствовать может только один каталог. Как только каталог определён, мы можем извлечь из имени каталога параметры сборки (например, оптимизированная, отладочная или профилировочная). Мы получаем последний компонент имени каталога, преобразуя набор слов, разделённых запятой, в набор слов, разделённых слэшем, и извлекая последнее слово этого набора при помощи функции *notdir*. Извлечение названия целевой платформы осуществляется таким же способом.

8.5 Частичные рабочие копии

В по-настоящему больших проектах простое создание рабочей копии и поддержка исходного кода может быть тяжким бременем для разработчиков. Если система состоит из большого числа модулей, и каждый разработчик работает над небольшой её частью, создание полной рабочей копии и компиляция всего проекта может быть непозволительной тратой времени. Вместо этого можно использовать централизованные справочные ночные сборки, служащие базой для заполнения недостающих файлов в деревьях каталогов исходных и бинарных файлов разработчиков.

Реализация этого функционала потребует осуществления двух типов поиска. Во-первых, если компилятору недостаёт заголовочного файла, нужно дать ему инструкцию искать этот файл в справочном дереве каталогов исходных файлов. Во-вторых, если *makefile*'у требуется какая-то библиотека, нужно дать ему инструкцию искать её в справочном дереве каталогов бинарных файлов. Для того, чтобы помочь компилятору найти недостающий исходный код, мы можем просто указать дополнительную опцию `-I` после аналогичной опции, специфицирующей локальные каталоги заголовочных файлов. Чтобы помочь *make* найти библиотеки, мы можем указать дополнительные каталоги в директиве `vpath`.

```
SOURCE_DIR     := ../mp3_player
REF_SOURCE_DIR := /reftree/src/mp3_player
REF_BINARY_DIR := /binaries/mp3_player
...
include_dirs := lib $(SOURCE_DIR)/lib $(SOURCE_DIR)/include
CPPFLAGS     += $(addprefix -I ,$(include_dirs))                    \
                $(addprefix -I $(REF_SOURCE_DIR)/,$(include_dirs))
vpath %.h       $(include_dirs)                                     \
                $(addprefix $(REF_SOURCE_DIR)/,$(include_dirs))

vpath %.a       $(addprefix $(REF_BINARY_DIR)/lib/, codec db ui)
```

Использование этого подхода предполагает, что наименьшей единицей, которую можно извлечь из репозитория CVS, является библиотека или программный модуль. В этом случае *make* сможет пропустить недостающие библиотеки и каталоги, если разработчик решил не делать их рабочих копий. Когда будет нужно использовать эти библиотеки, спецификация пути поиска поможет автоматически заполнить недостающие файлы.

Переменная `modules` нашего *makefile*'а содержит список подкаталогов, в которых следует осуществлять поиск файлов *module.mk*. Если эти подкаталоги не содержатся в рабочей копии, нужно удалить эти подкаталоги из списка. Кроме того, можно присваивать значение переменной `modules` при помощи функции *wildcard*:

```
modules := $(dir $(wildcard lib/*/module.mk))
```

Это выражение вернёт список всех каталогов, содержащих файл *module.mk*. Заметьте, что благодаря использованию функции *dir* имя каждого каталога будет оканчиваться слэшем.

make также может осуществлять поддержку создания частичных рабочих копий на уровне отдельных файлов, при сборке библиотеки соединяя объектные файлы из локальной копии разработчика и, в случае необходимости, из справочного дерева каталогов. Однако этот подход имеет множество недостатков и, судя по моему опыту, разработчикам он приносит больше вреда, чем пользы.

8.6 Справочные сборки, библиотеки и инсталляторы

Мы уже рассмотрели все средства, необходимые для реализации справочных сборок. Настройка головного *makefile*'а для добавления этого функционала не займёт много усилий. Просто заменим простые присваивания значений переменных `SOURCE_DIR` и `BINARY_DIR` условными (`?=`). Сценарий, выполняемый программой *cron*, может быть построен с использованием следующего подхода:

1. Перенаправить поток вывода и определить имена файлов журналов.
2. Очистить каталоги старых сборок и удалить лишние файлы из справочного дерева каталогов исходных файлов.
3. Сделать рабочую копию свежей версии исходного кода.
4. Определить переменные, отвечающие за расположение исходных и бинарных файлов.
5. Осуществить вызов *make*.
6. Проверить, содержат ли файлы журналов ошибки.
7. Составить файл символов (TAGS-файл), и, при необходимости, обновить базу данных файлов (locate database)[2].
8. Послать письмо с отчётом об успешном (или неудачном) завершении сборки.

В модели разработки с применением справочных сборок удобно поддерживать несколько старых сборок на случай, если чьё-то злонамеренное вмешательство повредит дерево каталогов. Я обычно храню 7 или 14 ночных сборок. Разумеется, сценарий, осуществляющий ночные сборки, описывает свои действия в файле журнала и удаляет устаревшие сборки и файлы журналов. Поиск ошибок в файле журнала обычно осуществляется при помощи сценария *awk*. Наконец, я использую сценарий, обновляющий файл *latest*, являющийся символической ссылкой на последнюю сборку. Для определения успешности сборки я включаю в каждый *makefile* цель `validate`. Сценарий, ассоциированный с этой целью, осуществляет проверку наличия всех необходимых целевых файлов:

```
.PHONY: validate_build
validate_build:
    test $(foreach f,$(RELEASE_FILES),-s $f -a) -e .
```

[2]База данных файлов — это совокупность всех имён файлов, существующих в файловой системе. Использование такой базы позволяет быстро находить файлы по имени. Трудно переоценить полезность такой базы при управлении большими проектами. Я предпочитаю реализовывать автоматическое обновление этой базы после ночной сборки.

Этот сценарий проверяет, что файлы, которые должны появиться в результате сборки, существуют и не пусты. Разумеется, такой подход никогда не заменит тестирования, однако он является удобной базовой проверкой целостности сборки. Если этот тест не пройден, *make* завершается с ошибкой, и сценарий, осуществляющий ночную сборку, сохраняет ссылку *latest* указывающей на предыдущую сборку.

Сторонние библиотеки всегда немного мешают управлению проектом. Я согласен с распространённым убеждением, что хранение больших бинарных файлов в CVS является не лучшей идеей. Причина кроется в том, что CVS не может хранить отличия бинарных файлов с помощью дельта-кодирования, в результате чего файлы системы RCS, лежащей в основе CVS, могут вырасти до огромных размеров. Хранение файлов больших размеров в репозитории CVS существенно замедляет многие базовые операции CVS, что сказывается на всём цикле разработки.

Если библиотеки сторонних разработчиков не хранятся в CVS, нужно управлять ими каким-то другим способом. Моим предложением является создание в справочном дереве каталога библиотек и включение версии каждой библиотеки в имя соответствующего каталога, как показано на рисунке 8.1.

Имена этих каталогов могут использоваться в *makefile*'е:

```
ORACLE_9011_DIR ?= /reftree/third_party/oracle-9.0.1.1/Ora90
ORACLE_9011_JAR ?= $(ORACLE_9011_DIR)/jdbc/lib/classes12.jar
```

Когда поставщик изменит версию своей библиотеки, создайте новый каталог в справочном дереве и объявите новую переменную в *makefile*'е. При использовании этого подхода *makefile*, должным образом поддерживаемый при помощи меток и ветвей системы контроля версий, всегда будет явно отражать версию используемой библиотеки.

Создание и поддержка инсталляторов также является сложной проблемой. Я уверен, что отделение базового процесса сборки от процесса создания инсталлятора является правильным решением. Инструменты для создания инсталляторов, существующие на момент написания этой книги, сложны и неустойчивы. Соединение этих инструментов с системой сборки (часто также являющейся сложной и неустойчивой) породит чрезвычайно сложную в поддержке систему. Вместо этого

```
reftree
‘--third-party
   |--oracle-8.0.7sp2
   ‘--oracle-9.0.1.1
```

Рис. 8.1: Структура каталогов, используемая для управления библиотеками сторонних разработчиков.

сценарий базовой сборки может помещать бинарные файлы в каталог продукта, содержащий все (или почти все) данные, необходимые инструменту создания инсталляторов. Управления этим инструментом может осуществляться при помощи собственного *makefile*'а, в конечном счёте производящего исполняемый файл установки.

Глава 9

Java

Многие Java-разработчики предпочитают использовать интегрированные среды разработки (Integrated Development Environments, IDE), например, Eclipse. У вас может возникнуть вопрос, зачем вам нужно использовать *make* в Java проектах, если есть такие известные альтернативы, как Ant и среды разработки Java? Эта глава содержит исследование значения *make* в среде Java, в частности, в ней приводится универсальный *makefile*, который может быть помещён с минимальными модификациями практически в любой Java-проект для осуществления всех стандартных задач сборки.

Использование *make* в совокупности с Java поднимает несколько проблем и предоставляет некоторые дополнительные возможности. Причиной этого является сочетание трёх основных факторов: во-первых, компилятор Java работает очень быстро; во-вторых, стандартный компилятор Java поддерживает синтаксис `@filename` для чтения параметров командной строки из файла; в третьих, если в коде Java-класса указан пакет, путь к *.class*-файлу определяется однозначно.

Стандартный компилятор Java работает очень быстро. Главной причиной этого является принцип работы директивы `import`. Подобно директиве `#include` препроцессора языка C, эта директива используется для обеспечения доступа к внешним символам. Однако вместо повторного чтения исходного кода, который затем потребует повторного разбора и анализа, компилятор Java считывает файлы классов напрямую. Поскольку символы, определяемые в файле класса, не могут измениться в процессе компиляции, компилятор производит кэширование классов. Даже в случае проектов среднего размера это означает, что компилятор Java избавлен от необходимости повторно считывать, разбирать и анализировать буквально миллионы строк кода, с которыми пришлось бы работать компилятору языка C. Менее существенный прирост производительности достигается за счёт сведéния к минимуму оптимизаций, выполняемых большинством компиляторов Java. Вместо статической оптимизации предпочтение отдаётся сложным оптимизациям времени

выполнения (just-in-time, JIT), осуществляемым виртуальной машиной Java (Java virtual machine, JVM).

Практически все крупные Java-проекты интенсивно используют *пакеты* (*packages*). Каждый класс инкапсулируется в пакет, определяющий область видимости символов, определённых в файле. Имена пакетов имеют иерархическую структуру и неявно определяют структуру файловой системы, предназначенную для их хранения. Например, пакет `a.b.c` неявно определяет структуру каталогов *a/b/c*. Код, объявленный соответствующей директивой как принадлежащий пакету `a.b.c`, будет скомпилирован в файлы классов и помещён в каталог *a/b/c*. Это означает, что обычный алгоритм *make*, отвечающий за ассоциацию бинарных файлов с соответствующими исходными файлами, не будет работать правильно. Однако это также означает, что нам больше не нужно указывать опцию `-o` для спецификации каталога, предназначенного для размещения объектного файла. Достаточно указать корень дерева каталогов бинарных файлов, одинаковый для всех исходных файлов. Это, в свою очередь, означает, что исходный код из различных каталогов может быть скомпилирован одной и той же командой.

Все стандартные компиляторы Java поддерживают синтаксис `@filename`, позволяющий считывать параметры командной строки из файла. Это имеет большое значение в сочетании с функционалом пакетов, поскольку позволяет производить компиляцию всего исходного кода единственным вызовом компилятора. Такой подход даёт значительный выигрыш в производительности, так как время, требуемое для загрузки и работы компилятора, является значительной частью времени выполнения сборки.

Итак, после составления соответствующей командной строки, компиляция 400 000 строк Java-кода занимает около трёх минут при использовании процессора Pentium 4 (2,5ГГц). Компиляция эквивалентного по размеру приложения, написанного на C++, потребует нескольких часов.

9.1 Альтернативы *make*

Как уже было замечено, сообщество Java-разработчиков с энтузиазмом принимает новые технологии. Рассмотрим две из них, имеющие отношение к *make* — *Ant* и интегрированные среды разработки.

9.1.1 Ant

Сообщество Java-разработчиков очень активно и производит новые инструменты с впечатляющей скоростью. Одним из таких инструментов является *Ant* — система сборки, призванная занять место *make* в процессе разработки Java-приложений. Как и *make*, *Ant* использует файл спецификации для определения целей и

реквизитов проекта. В отличие от *make*, *Ant* написан на языке Java и принимает файлы спецификации в формате XML.

Чтобы дать вам представление о файле спецификации в формате XML, приведу небольшую выдержку из файла сборки для *Ant*:

```
<target name="build"
        depends="prepare, check_for_optional_packages"
        description="--> compiles the source code">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${build.classes}"/>
  <mkdir dir="${build.lib}"/>

  <javac srcdir="${java.dir}"
         destdir="${build.classes}"
         debug="${debug}"
         deprecation="${deprecation}"
         target="${javac.target}"
         optimize="${optimize}" >
    <classpath refid="classpath"/>
  </javac>

  ...

  <copy todir="${build.classes}">
    <fileset dir="${java.dir}">
      <include name="**/*.properties"/>
      <include name="**/*.dtd"/>
    </fileset>
  </copy>
</target>
```

Как вы могли заметить, цель объявляется при помощи XML тега `<target>`. Каждая цель имеет имя и список зависимостей, указанных в атрибутах `name` и `depends` соответственно. Действия, выполняемые *Ant*, называются *задачами* (*tasks*). Задачи реализованы на языке Java и привязаны к XML тегу. Например, задача создания каталога специфицируется при помощи тега `<mkdir>` и вызывает выполнение метода `Mkdir.execute`, который в конечном итоге вызывает метод `File.mkdir`. Насколько это возможно, все задачи реализуются средствами Java API.

Эквивалентный файл сборки *make* содержит следующий код:

```
# производит компиляцию исходного кода
build: $(all_javas) prepare check_for_optional_packages
    $(MKDIR) -p $(build.dir) $(build.classes) $(build.lib)
    $(JAVAC) -sourcepath $(java.dir) \
             -d $(build.classes)     \
             $(debug)                \
             $(deprecation)          \
             -target $(javac.target) \
```

```
        $(optimize)             \
        -classpath $(classpath) \
        @$<
...
$(FIND) . \( -name '*.properties' -o -name '*.dtd' \) | \
$(TAR) -c -f - -T - | $(TAR) -C $(build.classes) -x -f -
```

Отрывок кода, приведённый выше, использует техники, которые мы ещё не обсуждали. Пока удовлетворимся тем, что реквизит `all.javac` содержит список всех *java* файлов, которые нужно скомпилировать. Задачи *Ant* `<mkdir>`, `<javac>` и `<copy>` также осуществляют проверку зависимостей. К примеру, если каталог уже существует, задача `mkdir` не выполнит никаких действий. Более того, если файлы Java-классов имеют более позднюю дату модификации, чем соответствующие исходные файлы, компиляция не будет осуществляться. Тем не менее, командный сценарий *make* осуществляет по существу такие же функции. *Ant* включает общую задачу, именуемую `<exec>`, используемую для запуска локальных программ.

Ant использует искусный и оригинальный подход, однако, при его использовании возникает несколько проблем, которые стоит рассмотреть:

- Несмотря на то, что *Ant* получил широкое распространение в Java-сообществе, вне сообщества *Ant* практически не распространён. К тому же, сомнительно, что его популярность когда-нибудь выйдет за пределы Java-проектов (по причинам, перечисленным далее). *make*, в свою очередь, успешно применяется во многих областях, включая разработку программного обеспечения, обработку документов и типографское дело, поддержку веб-сайтов. Понимание *make* очень важно для любого, кому требуется работать в различных программных системах.

- Выбор XML как языка спецификаций вполне разумен для Java-приложения. Однако читать и писать спецификации на языке XML большинству людей не очень удобно. Хороший XML-редактор может быть нелегко найти или интегрировать с существующими инструментами (если моя интегрированная среда разработки не содержит хорошего XML-редактора, мне придётся либо менять среду разработки, либо искать такой редактор и использовать его отдельно). Как вы могли видеть из предыдущего примера, *Ant*-диалект XML довольно избыточен по сравнению с синтаксисом *make*, и полон специфических для XML особенностей.

- В процессе работы с файлами *Ant* вам нужно преодолевать некоторую косвенность ваших спецификаций. Задача *Ant* `<mkdir>` те вызывает соответствующую программу *mkdir* вашей системы. Вместо этого вызывается метод `mkdir()` класса `java.io.File`. Результатом вызова может быть совсем не то,

что вы ожидаете. По существу, любое предположение программиста о поведении основных инструментов *Ant* должно быть проверено с привлечением документации по *Ant* или Java, либо исходного кода *Ant*. В добавок, для вызова, к примеру, компилятора Java, вам может понадобиться разобраться в использовании десятка или более незнакомых XML атрибутов, например, `srcdir`, `debug` и т.д., не вошедших в руководство пользователя компилятора. В противоположность этому *make* совершенно прозрачен; как правило, вы можете просто набирать команды прямо в интерпретаторе и следить за их поведением.

- И всё же, несомненно, *Ant* переносим, как и *make*. Как показано в главе 7, написание переносимых *makefile*'ов, как и написание переносимых спецификаций *Ant*, требуют опыта и особых знаний. Программисты писали переносимые *makefile*'ы два десятилетия. Более того, в документации *Ant* отмечается, что *Ant* имеет проблемы переносимости, связанные с символическими ссылками Unix и длинными именами файлов в Windows, а MacOS X является единственной операционной системой Apple, поддерживаемой *Ant*, поддержка же других платформ не гарантируется. К тому же, базовые операции наподобие выставления флага исполняемости файлов не могут осуществляться при помощи Java API, для этого требуется вызов внешней программы. Переносимость никогда не может быть простой или полной.

- Программа *Ant* не предоставляет подробного отчёта о своих действиях. Поскольку задачи *Ant* реализованы не в виде командных сценариев, отображение действий, совершаемых этими задачами, вызывает определённые трудности. Как правило, вывод состоит из выражений на естественном языке, выдаваемых выражениями `print`, добавленными автором задачи. Эти выражения не могут быть выполнены пользователем в командной строке. В противоположность этому, строки текста, отображаемые *make* являются выражениями интерпретатора и могут быть копированы из вывода и вставлены в командный интерпретатор для повторного выполнения. Это означает, что *Ant* менее полезен для разработчиков, пытающихся понять процесс сборки и способ работы инструментов, используемых в этом процессе. Кроме того, это не даёт разработчику возможности повторно использовать элементы этих задач экспромтом, при помощи клавиатуры.

- Последняя и наиболее важная проблема заключается в том, что *Ant* сдвигает парадигмы осуществления сборок, призывая использовать компилируемый язык программирования взамен интерпретируемого. Задачи *Ant* написаны на языке Java. Если какая-то задача не реализована или делает не то, что вы хотите, вам нужно либо реализовать собственную задачу на Java, либо использовать задачу `<exec>` (разумеется, если вам приходится часто исполь-

зовать задачу `<exec>`, то гораздо проще использовать *make* с его макросами, функциями и более компактным синтаксисом).

С другой стороны, интерпретируемые языки программирования были изобретены для решения именно таких проблем. *make* существует около тридцати лет и может быть использован в большинстве сложных ситуаций без расширения своей реализации. Разумеется, за эти тридцать лет была реализована поддержка множества новых возможностей. Многие из них задуманы и реализованы в GNU *make*.

Ant является замечательной программой, широко распространённой в Java-сообществе. Тем не менее, прежде, чем приступить к новому проекту, тщательно убедитесь, что *Ant* является подходящим инструментом для вашей среды разработки. Надеюсь, эта глава докажет вам, что *make* может быть успешно использован для осуществления сборки вашего Java-проекта.

9.1.2 Интегрированные среды разработки

Многие Java-разработчики используют интегрированные среды разработки, совмещающие в единой (как правило, графической) среде редактор, компилятор, отладчик и инструмент для навигации по исходному коду. В качестве примеров можно привести такие проекты с открытым исходным кодом, как Eclipse (*http://www.eclipse.org*) и Emacs JDEE (*http://jdee.sunsite.dk*), а также, если рассматривать коммерческие разработки, Sun Java Studio (*http://www.sun.com/software/sundev/jde*) и JBuilder (*http://www.borland.com/jbuilder*). Эти среды, как правило, имеют понятие процесса сборки проекта, заключающегося в компиляции необходимых файлов и запуска приложения на выполнение.

Если интегрированная среда разработки поддерживает все эти операции, зачем тогда нам рассматривать использование *make*? Наиболее очевидной причиной является переносимость. Если возникнет необходимость осуществить сборку проекта на другой платформе, сборка может закончится неудачей. Несмотря на то, что код Java сам по себе является переносимым, инструменты для работы с ним, как правило, таковыми не являются. Например, конфигурационные файлы вашего проекта могут включать списки путей в стиле Unix или Windows, это может стать причиной ошибки при попытке запуска сборки под управлением другой операционной системы. Второй причиной является тот факт, что *make* поддерживает автоматические сборки. Некоторые интегрированные среды разработки поддерживают пакетные сборки, а некоторые нет. Качество этой поддержки также варьируется. Наконец, встроенная поддержка сборок часто бывает довольно ограниченной. Если вы хотите реализовать собственную структуру каталогов, соответствующую структуре релизов вашего проекта, интегрировать файлы помощи внешних приложений, поддерживать автоматическое тестирование, ветвление и параллельные

треки разработки, скорее всего, вы обнаружите, что встроенная поддержка сборок не подходит для ваших нужд.

По собственному опыту я могу судить, что интегрированные среды разработки вполне подходят для небольших немасштабируемых приложений, однако промышленные системы сборки требуют большей поддержки, и *make* может её обеспечить. Обычно я использую интегрированную среду разработки для написания и отладки кода и составляю *makefile* для промышленных сборок и релизов. Во время разработки я использую интегрированную среду для компиляции проекта в состояние, пригодное для отладки. Однако если я изменяю много файлов или модифицирую файлы, являющиеся входными файлами для генератора кода, я запускаю *makefile*. Интегрированная среда разработки, которую я использовал, не имела соответствующей поддержки внешних программ, осуществляющих генерацию кода. Обычно сборки, полученные с помощью интегрированной среды, не подходят для поставок внутренним или внешним потребителям. Для таких задач я использую *make*.

9.2 Универсальный *makefile* для Java

Следующий пример демонстрирует универсальный *makefile* для сборки Java-проектов. Я объясню каждую из его частей далее в этой главе.

```
# Общий makefile для Java-проекта.
VERSION_NUMBER := 1.0

# Определения базовых каталогов
SOURCE_DIR     := src
OUTPUT_DIR     := classes

# Инструменты Unix
AWK            := awk
FIND           := /bin/find
MKDIR          := mkdir -p
RM             := rm -rf
SHELL          := /bin/bash

# Пути для поддержки работы программ
JAVA_HOME      := /opt/j2sdk1.4.2_03
AXIS_HOME      := /opt/axis-1_1
TOMCAT_HOME    := /opt/jakarta-tomcat-5.0.18
XERCES_HOME    := /opt/xerces-1_4_4
JUNIT_HOME     := /opt/junit3.8.1

# Инструменты Java
JAVA           := $(JAVA_HOME)/bin/java
JAVAC          := $(JAVA_HOME)/bin/javac
```

```
JFLAGS         := -sourcepath $(SOURCE_DIR)  \
                  -d $(OUTPUT_DIR)           \
                  -source 1.4

JVMFLAGS       := -ea                        \
                  -esa                       \
                  -Xfuture

JVM            := $(JAVA) $(JVMFLAGS)

JAR            := $(JAVA_HOME)/bin/jar
JARFLAGS       := cf

JAVADOC        := $(JAVA_HOME)/bin/javadoc
JDFLAGS        := -sourcepath $(SOURCE_DIR) \
                  -d $(OUTPUT_DIR)          \
                  -link http://java.sun.com/products/jdk/1.4/docs/api

# Jar архивы
COMMONS_LOGGING_JAR := $(AXIS_HOME)/lib/commons-logging.jar

LOG4J_JAR           := $(AXIS_HOME)/lib/log4j-1.2.8.jar
XERCES_JAR          := $(XERCES_HOME)/xerces.jar
JUNIT_JAR           := $(JUNIT_HOME)/junit.jar

# Определяем путь к классам Java
class_path := OUTPUT_DIR          \
              XERCES_JAR          \
              COMMONS_LOGGING_JAR \
              LOG4J_JAR           \
              JUNIT_JAR

# Пробел
space := $(empty) $(empty)

# $(call build-classpath, variable-list)
define build-classpath
  $(strip                                          \
    $(patsubst :%,%,                               \
      $(subst : ,:,                                \
        $(strip                                    \
          $(foreach j,$1,$(call get-file,$j):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                         \
    $($1)                                         \
      $(if $(call file-exists-eval,$1),,          \
        $(warning Файл, указанный в переменной    \
```

```
                ’$1’ ($($1)), не найден)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                       \
    $(if $($1),,$(warning ’$1’ has no value))  \
    $(wildcard $($1)))
endef

# $(call brief-help, makefile)
define brief-help
  $(AWK) ’$$1 ~ /^[^.][-A-Za-z0-9]*:/                    \
         { print substr($$1, 1, length($$1)-1) }’ $1 | \
  sort |                                                 \
  pr -T -w 80 -4
endef

# $(call file-exists, wildcard-pattern)
file-exists = $(wildcard $1)

# $(call check-file, file-list)
define check-file
  $(foreach f, $1,                          \
    $(if $(call file-exists, $($f)),,    \
      $(warning $f ($($f)) is missing)))
endef

# $(call make-temp-dir, root-opt)
define make-temp-dir
  mktemp -t $(if $1,$1,make).XXXXXXXXXX
endef

# MANIFEST_TEMPLATE - шаблон файла манифеста, предназначенный
#                     для обработки макропроцессором m4
MANIFEST_TEMPLATE := src/manifest/manifest.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf

# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"            \
     --define=IMPL_VERSION=$(VERSION_NUMBER) \
     --define=SPEC_VERSION=$(VERSION_NUMBER) \
     $(if $3,$3,$(MANIFEST_TEMPLATE))        \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
```

```
endef

# Определяем переменную CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))

# make-directories - убеждаемся, что выходной каталог существует
make-directories := $(shell $(MKDIR) $(OUTPUT_DIR))

# help - цель по умолчанию
.PHONY: help
help:
    @$(call brief-help, $(CURDIR)/Makefile)

# all - осуществляет полную сборки системы
.PHONY: all
all: compile jars javadoc

# all_javas - временный файл для хранения списка исходных файлов
all_javas := $(OUTPUT_DIR)/all.javas

# compile - компилирует исходный код
.PHONY: compile
compile: $(all_javas)
    $(JAVAC) $(JFLAGS) @$<

# all_javas - составляет список исходных файлов
.INTERMEDIATE: $(all_javas)
$(all_javas):
    $(FIND) $(SOURCE_DIR) -name '*.java' > $@

# jar_list - список всех jar-архивов
jar_list := server_jar ui_jar

# jars - создаёт все jar-архивы
.PHONY: jars
jars: $(jar_list)

# server_jar - создаёт архив $(server_jar)
server_jar_name     := $(OUTPUT_DIR)/lib/a.jar
server_jar_manifest := src/com/company/manifest/foo.mf
server_jar_packages := com/company/m com/company/n

# ui_jar - создаёт архив $(ui_jar)
ui_jar_name     := $(OUTPUT_DIR)/lib/b.jar
ui_jar_manifest := src/com/company/manifest/bar.mf
ui_jar_packages := com/company/o com/company/p

# Создаёт явные правила для каждого архива
# $(foreach j, $(jar_list), $(eval $(call make-jar,$j)))
$(eval $(call make-jar,server_jar))
```

```
$(eval $(call make-jar,ui_jar))

# javadoc - создаёт документацию Java doc
.PHONY: javadoc
javadoc: $(all_javas)
    $(JAVADOC) $(JDFLAGS) @$<

.PHONY: clean
clean:
    $(RM) $(OUTPUT_DIR)

.PHONY: classpath
classpath:
    @echo CLASSPATH='$(CLASSPATH)'

.PHONY: check-config
check-config:
    @echo Проверяем конфигурацию...
    $(call check-file, $(class_path) JAVA_HOME)

.PHONY: print
print:
    $(foreach v, $(V), \
      $(warning $v = $($v)))
```

9.3 Компиляция Java кода

Есть два способа компиляции кода Java с помощью *make*: традиционный подход, вызывающий *javac* для компиляции каждого файла, и более быстрый подход, изложенный ранее и использующий синтаксис `@filename`.

Быстрый подход: компиляция всех исходных файлов за один раз

Давайте более детально рассмотрим быстрый подход. Обратите внимание на следующий фрагмент универсального *makefile*'а:

```
# all_javas - временный файл для хранения списка исходных файлов
all_javas := $(OUTPUT_DIR)/all.javas

# compile - компилирует исходный код
.PHONY: compile
compile: $(all_javas)
    $(JAVAC) $(JFLAGS) @$<

# all_javas - составляет список исходных файлов
.INTERMEDIATE: $(all_javas)
$(all_javas):
    $(FIND) $(SOURCE_DIR) -name '*.java' > $@
```

Абстрактная цель `compile` вызывает *javac* для компиляции всего исходного кода проекта.

Реквизит `$(all_java)` — это файл, *all.javas*, содержащий список исходных файлов Java, по одному файлу на каждой строке. Вовсе необязательно размещать каждый файл на отдельной строке, однако так гораздо легче производить фильтрацию этого списка командой `grep -v`, если в этом возникнет необходимость. Правило создания файла *all.javas* помечено как `.INTERMEDIATE`, поэтому *make* будет удалять этот файл после каждого запуска и создавать его заново перед каждой компиляцией. Командный сценарий для создания файла очень прост. Для обеспечения максимальной переносимости мы используем команду *find* для извлечения всех исходных файлов Java из дерева каталогов с исходными файлами. Эта команда работает не очень быстро, однако мы можем быть уверены в её корректной работе. Более того, при изменении структуры дерева каталогов с исходным кодом нам практически не придётся вносить изменений в этот командный сценарий.

Если список каталогов, содержащих исходный код, определён и может быть указан в вашем *makefile*'е, вы можете использовать более производительный способ составления файла *all.javas*. Если список каталогов с исходным кодом не очень велик и помещается в командной строке, не нарушая ограничений, накладываемых операционной системой, можно использовать следующий сценарий:

```
$(all_javas):
    shopt -s nullglob; \
    printf "%s\n" $(addsuffix /*.java,$(PACKAGE_DIRS)) > $@
```

Этот сценарий использует шаблоны командного интерпретатора для определения списка Java-файлов в каждом каталоге. Однако если каталог не содержит Java-файлов, нам хотелось бы, чтобы раскрытие шаблон порождало пустую строку, а не текст исходного шаблона (именно таково поведение по умолчанию многих командных интерпретаторов). Для достижения этого эффекта используется опция командного интерпретатора *bash* `shopt -s nullglob`. Большинство других интерпретаторов имеет подобную опцию. Наконец, мы используем шаблоны и команду `printf` вместо `ln -l`, поскольку эти инструменты интегрированы в *bash*, поэтому потребуется выполнение всего одной программы независимо от числа каталогов.

Мы можем избежать использования шаблонов интерпретатора при помощи вызова функции *wildcard*:

```
$(all_javas):
    print "%s\n" $(wildcard \
                  $(addsuffix /*.java,$(PACKAGE_DIRS))) > $@
```

Если ваш проект содержит много каталогов с исходным кодом (или пути к ним имеют очень большую длину), предыдущий сценарий может превысить предел длины командной строки вашей системы. В этом случае более предпочтительным является следующий вариант:

```
.INTERMEDIATE: $(all_javas)
$(all_javas):
    shopt -s nullglob;           \
    for f in $(PACKAGE_DIRS);    \
    do                           \
      printf "%s\n" $$f/*.java;  \
    done > $@
```

Заметим, что цель `compile` и вспомогательное правило следуют подходу, основанному на нерекурсивном вызове *make*. Не важно, сколько подкаталогов в нашем проекте, мы используем единственный *makefile* и производим единственный вызов компилятора. Если вам нужно произвести компиляцию всего исходного кода, этот подход является наиболее быстрым.

К тому же, мы совершенно не используем информацию о зависимостях. Используя эти правила, *make* не знает о связях между файлами и не заботится о датах их модификации. Он просто осуществляет компиляцию всего исходного кода при каждом вызове. В качестве бонуса мы получаем возможность вызывать *make* из каталога с исходными, а не бинарными файлами. В контексте возможностей управления зависимостями *make* это может выглядеть как неразумный способ организации *makefile*'а, однако давайте примем во внимание следующие доводы:

- Альтернатива (краткий обзор которой мы произведём позже) использует стандартный подход, основанный на зависимостях. При этом для каждого файла создаётся новый процесс *javac*, что увеличивает накладные расходы. Однако если наш проект не очень велик, компиляция всех исходных файлов займёт не намного больше времени, чем компиляция нескольких файлов, поскольку компилятор *javac* работает очень быстро, а создание новых процессов происходит относительно медленно. Любая сборка, занимающая менее 15 секунд, практически эквивалентна другой такой же, независимо от количества работы, которую необходимо выполнить. Например, компиляция приблизительно пятисот исходных файлов (дистрибутива *Ant*) занимает 14 секунд при выполнении на моём Pentium 4 1.8 ГГц, имеющем 512 Мб оперативной памяти. Компиляция одного файла занимает пять секунд.

- Бóльшая часть разработчиков будет использовать некий аналог рабочей среды, предоставляющей быструю компиляцию отдельных файлов. *makefile* же в основном будет использоваться в том случае, если изменения охватывают большой участок кода, или требуется чистая сборка, или же сборка осуществляется без вмешательства человека.

- Как мы увидим, усилия, требуемые для реализации и поддержки подхода, основанного на зависимостях, сравнимы усилиями, необходимыми для реализации разделения деревьев каталогов исходных и бинарных файлов для

проектов, написанных на C/C++(эта тема обсуждается в главе 8). Эту задачу не стоит недооценивать.

Как мы увидим в следующих примерах, переменная `PACKAGE_DIRS` используется не только для составления файла *all.javas*. Поддержка корректного значения этой переменной может быть трудоёмким и потенциально сложным шагом. В случае небольших проектов список каталогов может указываться явно прямо в *makefile*'е, однако при росте числа каталогов до нескольких сотен ручное редактирование этого списка становится довольно неприятным занятием может привести к ошибкам. Более благоразумным способом может быть использование программы *find* для поиска соответствующих каталогов:

```
# $(call find-compilation-dirs, root-directory)
  find-compilation-dirs =                          \
    $(patsubst %/,%,                               \
      $(sort                                       \
        $(dir                                      \
          $(shell $(FIND) $1 -name '*.java'))))
  PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

Команда `find` возвращает список файлов, функция *dir* отсекает лишнюю часть имени, оставляя только имя каталога, функция *sort* удаляет из списка дубликаты, а функция *patsubst* удаляет слэш на конце каждого имени. Обратите внимание на то, что функция *find-compilation-dirs* находит все файлы, подлежащие компиляции, только для того, чтобы отсечь имена файлов, в то время как правило, ассоциированное с *all.javas* использует шаблоны для восстановления этих имён. Это может показаться напрасным расточительством ресурсов, однако я часто замечаю, что наличие списка пакетов, содержащих исходный код, чрезвычайно удобно в других аспектах сборки, например, при сканировании конфигурационных файлов EJB. Если в вашем случае список пакетов не требуется, просто используйте один из более простых методов, упомянутых при обсуждении составления файла *all.javas*.

Компиляция с учётом зависимостей

Для реализации сборки с полным учётом зависимостей нам потребуется инструмент для извлечения информации о зависимостях из исходных файлов Java, подобный команде `cc -M`. Программа Jikes (*http://www.ibm.com/developerworks/opensource/jikes*) — это компилятор Java с открытым исходным кодом, поддерживающий эту возможность при использовании опций `-makefile` или `+M`. Jikes — не идеальный инструмент для разделения исходного кода и бинарных файлов, потому что он всегда записывает файл зависимостей в тот же каталог, в котором

находится исходный файл, однако он бесплатен и эффективен. Есть и положительная сторона: файлы зависимостей создаются во время компиляции, поэтому дополнительный вызов компилятора не требуется.

Ниже приведён пример функции для работы с зависимостями и правила, использующего эту функцию:

```
%.class: %.java
    $(JAVAC) $(JFLAGS) +M $<
    $(call java-process-depend,$<,$@)

# $(call java-process-depend, source-file, object-file)
define java-process-depend
  $(SED) -e 's/^.*\.class *:/$2 $(subst .class,.d,$2):/' \
         $(subst .java,.u,$1) > $(subst .class,.tmp,$2)
  $(SED) -e 's/#.*//'                                    \
         -e 's/^[^:]*: *//'                              \
         -e 's/ *\\$$$$//'                               \
         -e '/^$$$$/ d'                                  \
         -e 's/$$$$/ :/' $(subst .class,.tmp,$2)         \
         >> $(subst .class,.tmp,$2)
  $(MV) $(subst .class,.tmp,$2).tmp $(subst .class,.d,$2)
endef
```

Этот сценарий требует, чтобы запуск *make* осуществлялся из каталога с бинарными файлами, и чтобы директива `vpath` указывала расположение исходных файлов. Если вы хотите использовать компилятор Jikes только для генерации зависимостей, обращаясь к другому компилятору для непосредственной генерации кода, вы можете использовать опцию `+B`, в этом случае Jikes не будет генерировать байткод.

В небольшом тесте производительности, в рамках которого происходила компиляция 223 Java-файлов, однострочная команда компиляции, описанная ранее, выполнялась на моей машине 9.9 секунд. Компиляция тех же 223 файлов с индивидуальным вызовом компилятора для каждого файла потребовала 411.6 секунд, т.е. в 41.5 раз больше времени. Более того, при использовании раздельной компиляции любая сборка, требующая компиляции более четырёх исходных файлов, будет занимать больше времени, чем компиляция всего проекта одной командой. Если генерация зависимостей и компиляция будут осуществляться разными программами, разница только увеличится.

Разумеется, среды разработки варьируются, однако всегда важно внимательно обдумать ваши цели. Минимизация числа файлов, подлежащих компиляции, не всегда будет означать минимизацию времени, требующегося для сборки системы. В случае языка Java полная проверка зависимостей и минимизация числа компилируемых файлов не являются необходимыми атрибутами хорошей среды программирования.

Определение переменной CLASSPATH

Одной из самых важных проблем при разработке программного обеспечения на языке Java является корректное определение переменной `CLASSPATH`. Эта переменная определяет, откуда будет загружаться код при разрешении ссылки на класс. Для корректной компиляции Java-приложения *makefile* должен включать правильное определение переменной `CLASSPATH`. Эта задача быстро становится сложной при добавлении Java-пакетов, программных интерфейсов приложений (API) и вспомогательных инструментов. Если правильное определение `CLASSPATH` может быть сложной задачей, имеет смысл делать эту задачу в каком-то одном месте.

Техника, которую я нашёл полезной, заключается в определении переменной `CLASSPATH` *makefile*'е для нужд не только *make*, но и других программ. Например, цель `classpath` может возвращать команду экспорта переменной `CLASSPATH` в среду командного интерпретатора, вызвавшего *make*:

```
.PHONY: classpath
classpath:
    @echo "export CLASSPATH='$(CLASSPATH)'"
```

Разработчики могут определять `CLASSPATH` следующим образом (если они используют *bash*):

```
$ eval $(make classpath)
```

Определить переменную `CLASSPATH` в среде Windows можно следующим образом:

```
.PHONY: windows_classpath
windows_classpath:
    regtool set /user/Environment/CLASSPATH \
        "$(subst /,\\,$(CLASSPATH))"
    control sysdm.cpl,@1,3 &
    @echo "Теперь нажмите кнопку <<Переменные окружения>>, " \
      "затем OK, затем снова OK."
```

Программа *regtool* является частью среды разработки Cygwin и предназначена для работы с реестром Windows. Однако простое обновление реестра не вызовет считывания нового значения. Одним из способов осуществления этой задачи является посещение диалогового окна «Переменные окружения» (Environment Variables) и закрытие этого окна при помощи кнопки OK.

Вторая строка сценария сообщает Windows, что нужно отобразить диалоговое окно «Свойства системы» (System Properties) и сделать активной вкладку «Дополнительно» (Advanced). К сожалению, командный сценарий не может отобразить

диалоговое окно «Переменные окружения» или активировать кнопку OK, поэтому последняя строка сценария предлагает пользователю завершить работу самостоятельно.

Экспорт переменной `CLASSPATH` в другие программы, такие как проектные файлы Emacs JDEE или JBuilder, осуществляется очень просто.

Непосредственное определение переменной `CLASSPATH` может также управляться при помощи *make*. Определение этой переменной очевидным способом определённо является разумной идеей:

```
CLASSPATH = /third_party/toplink-2.5/TopLink.jar:/third_party/...
```

Из соображений переносимости более предпочтительным способом является использование переменных:

```
# Определение Java classpath
class_path := OUTPUT_DIR          \
              XERCES_JAR          \
              COMMONS_LOGGING_JAR \
              LOG4J_JAR           \
              JUNIT_JAR
...
# Определение CLASSPATH
export CLASSPATH := $(call build-classpath, $(class_path))
```

(Определение `CLASSPATH`, приведённое в коде универсального *makefile*'а, более показательно и полезно). Должным образом реализованная функция *build-classpath* решает несколько раздражающих проблем:

- Очень просто собрать значение `CLASSPATH` из частей. Например, если используется несколько серверов приложений, может потребоваться изменение `CLASSPATH`. Различные версии `CLASSPATH` могут заключаться в секции `ifdef` и выбираться на основании значения какой-либо переменной *make*.

- Люди, занимающиеся поддержкой *makefile*'а, не должны волноваться о внутренних пробелах, символах новой строки или переносах строк, функция *build-classpath* осуществляет необходимые операции самостоятельно.

- Функция *build-classpath* может выбирать разделитель путей автоматически, делая тем самым значение переменной корректным для Windows и Unix.

- Функция *build-classpath* может осуществлять проверку правильности элементов списка путей. В частности, одной из раздражающих проблем *make* является то, что вычисление переменных, значение которых не определено, просто возвращает пустую строку. В большинстве случаев такое поведение полезно,

однако иногда оно может встать на вашем пути. В этом случае значение переменной `CLASSPATH` будет иметь фиктивное значение[1]. Мы можем решить эту проблему, добавив проверку определённости переменных, входящих в список путей, в функцию *build-classpath*. Функция также может проверять существование каждого файла или каталога, входящего в список, и, в случае невыполнения ограничений, выводить соответствующее предупреждение.

- Наконец, для реализации наиболее изощрённого функционала (например, обработки пробелов в именах файлов или путях поиска) может быть удобно использовать триггер для обработки переменной `CLASSPATH`.

Ниже приведена реализация функции *build-classpath*, учитывающая первые три пункта нашего списка:

```
# $(call build-classpath, variable-list)
define build-classpath
  $(strip                                          \
    $(patsubst %:,%,                               \
      $(subst : ,:,                                \
        $(strip                                    \
          $(foreach c,$1,$(call get-file,$c):)))))
endef

# $(call get-file, variable-name)
define get-file
  $(strip                                        \
    $($1)                                        \
    $(if $(call file-exists-eval,$1),,           \
      $(warning Файл, указанный в переменной     \
                '$1' ($($1)), не найден)))
endef

# $(call file-exists-eval, variable-name)
define file-exists-eval
  $(strip                                                  \
    $(if $($1),,$(warning Переменная'$1' не определена)) \
    $(wildcard $($1)))
endef
```

Функция *build-classpath* проходит по всем словам своего аргумента, производя проверку каждого элемента и соединяя эти элементы, используя разделитель путей (в нашем случае это `:`). Реализовать автоматический выбор разделителя путей теперь очень просто. Затем функция удаляет пробелы, добавленные функцией *get-file* и циклом *foreach*. Затем функция удаляет последний разделитель, добавленный

[1] Для обнаружения этой ситуации можно попробовать использовать опцию `--warn-undefined-variables`, однако это приведёт к предупреждениям, относящимся к тем переменным, неопределённое значение которых нас устраивает.

циклом *foreach*. Наконец, весь список подаётся на вход функции *strip*, благодаря чему удаляются лишние пробелы, добавленные продолжением строк.

Функция *get-file* принимает на вход имя переменной и осуществляет проверку существования файла, имя которого является значением переменной. Если файл не существует, генерируется предупреждение. Функция возвращает значение переменной независимо от того, существует ли соответствующий файл, так как это значение может быть полезным для пользователя. В некоторых случаях функция *get-file* может быть применена к файлу, который будет сгенерирован позже и пока не существует.

Последняя функция, *file-exists-eval*, принимает в качестве аргумента имя переменной, содержащей имя файла. Если переменная содержит пустую строку, генерируется предупреждение, в противном случае для проверки существования файла (или списка файлов) используется вызов функции *wildcard*

Если функция *build-classpath* применяется к фиктивным значениям, при запуске мы увидим следующие ошибки:

```
Makefile:37: Файл, указанный в переменной 'TOPLINKX_25_JAR'
             (/usr/java/toplink-2.5/TopLinkX.jar), не найден
...
Makefile:37: Переменная 'XERCES_142_JAR' не определена
Makefile:37: Файл, указанный в переменной 'XERCES_142_JAR'
             ( ), не найден
```

Этот пример демонстрирует существенный прогресс по сравнению с молчанием, которое мы получаем при использовании простого подхода.

Существование функции *get-file* подразумевает, что мы можем обобщить задачу поиска входных файлов.

```
# $(call get-jar, variable-name)
define get-jar
  $(strip                                                    \
    $(if $($1),,$(warning Переменная '$1' пуста))            \
    $(if $(JAR_PATH),,$(warning Переменная JAR_PATH пуста))  \
    $(foreach d, $(dir $($1)) $(JAR_PATH),                   \
      $(if $(wildcard $d/$(notdir $($1))),                   \
        $(if $(get-jar-return),,                             \
          $(eval get-jar-return := $d/$(notdir $($1))))))    \
    $(if $(get-jar-return),                                  \
      $(get-jar-return)                                      \
      $(eval get-jar-return :=),                             \
      $($1)                                                  \
      $(warning get-jar: Файл '$1' не найден в $(JAR_PATH))))
endef
```

Здесь мы определяем переменную `JAR_PATH`, содержащую пути для поиска файлов. Возвращается первый найденный файл. Параметром функции является имя

переменной, содержащей путь к архиву jar. Мы производим поиск jar-файла, используя сначала путь, содержащийся в переданной переменной, затем набор путей, содержащихся в переменной `JAR_PATH`. Чтобы реализовать такое поведение, список каталогов в цикле *foreach* составляется из значения переменной, за которым сделует значение переменной `JAR_PATH`. Два других обращения к параметру обрамляются вызовом функции *notdir*, благодаря чему имя архива можно соединить с соответствующим элементом списка. Обратите внимание на то, что мы не можем выйти из цикла *foreach*. Вместо этого, однако, мы используем функцию *eval* для определения переменной `get-jar-return`, используемую для хранения первого найденного файла. После выхода из цикла мы возвращаем значение временной переменной или генерируем предупреждение, если файл не был найден. Важно не забыть сбросить значение временной переменной перед завершением работы макроса.

Эта функция по существу является реализацией директивы `vpath` в контексте определения переменной `CLASSPATH`. Чтобы понять это, вспомним, что директива `vpath` используется *make* для нахождения реквизитов, которые не были найдены по их относительному пути. В такой ситуации *make* производит поиск реквизитов в каталогах, указанных директивой `vpath`, и подставляет дополненный путь в автоматические переменные `$^`, `$?` и `$+`. Мы хотим, чтобы для определения переменной `CLASSPATH` *make* производил поиск пути к каждому jar-файлу и производил конкатенацию переменной `CLASSPATH` с этим дополненным путём. Поскольку *make* не имеет встроенной поддержки этого функционала, мы добавляем его самостоятельно. Разумеется, вы можете просто всегда прописывать полные пути к jar-файлам, предоставив задачу поиска виртуальной машине Java, однако переменная `CLASSPATH` и без того быстро становится длинной. На некоторых операционных системах длина переменных окружения ограничена и существует опасность усечения длинных значений `CLASSPATH`. Например, на операционной системе Windows XP длина значения переменной окружения ограничена 1023 символами. В добавок, даже если переменая `CLASSPATH` не будет усечена, виртуальная машина Java должна производить поиск в `CLASSPATH` при загрузке классов, что замедляет работу приложения.

9.4 Управление архивами Java

Сборка и управление Java-архивами поднимают проблемы, отличные от тех, с которыми мы сталкивались при сборке библиотек C/C++. На это есть три причины. Во-первых, элементы Java-архива адресуются относительным путём, поэтому точные имена файлов, передаваемых программе *jar*, нужно тщательно контролировать. Во-вторых, в Java-сообществе есть тенденция соединять архивы, чтобы всё приложение могло размещаться в единственном архиве. Наконец, Java-архивы мо-

гут содержать файлы, отличные от файлов классов, например, файл манифеста, файлы свойств и XML-файлы.

Базовая команда для создания Java-архива при помощи GNU *make* выглядит следующим образом:

```
JAR      := jar
JARFLAGS := -cf

$(FOO_JAR): реквизиты...
    $(JAR) $(JARFLAGS) $@ $^
```

Программа *jar* может принимать вместо имён файлов имена каталогов, в этом случае в архив будет помещено всё содержимое указанных каталогов. Это может быть очень удобно, особенно при использовании совместно с опцией `-C`, временно изменяющей текущий каталог:

```
JAR      := jar
JARFLAGS := -cf

.PHONY: $(FOO_JAR)
$(FOO_JAR):
    $(JAR) $(JARFLAGS) $@ -C $(OUTPUT_DIR) com
```

Здесь файл архива объявлен абстрактной целью. Однако при повторном запуске *makefile*'а архив не будет создаваться заново, поскольку эта цель не имеет реквизитов. Как и в случае команды *ar*, описанной в одной из предыдущих глав, смысла в использовании флага обновления архива, `-u`, практически нет, поскольку эта операция занимает практически такое же (или даже большее) время, что и операция создания нового архива.

Java-архив часто включает файл манифеста, в котором указан поставщик, API и номер версии. Простой файл манифеста может выглядеть следующим образом:

```
Name: JAR_NAME
Specification-Title: SPEC_NAME
Implementation-Version: IMPL_VERSION
Specification-Vendor: Generic Innovative Company, Inc.
```

Этот файл содержит три переменных, `JAR_NAME`, `SPEC_NAME` и `IMPL_VERSION`, которые могут быть заменены реальными значениями при создании архива с помощью *sed*, *m4* или вашего любимого редактора потоков. Ниже приведена функция для обработки файла манифеста:

```
MANIFEST_TEMPLATE := src/manifests/default.mf
TMP_JAR_DIR       := $(call make-temp-dir)
TMP_MANIFEST      := $(TMP_JAR_DIR)/manifest.mf
```

```
# $(call add-manifest, jar, jar-name, manifest-file-opt)
define add-manifest
  $(RM) $(dir $(TMP_MANIFEST))
  $(MKDIR) $(dir $(TMP_MANIFEST))
  m4 --define=NAME="$(notdir $2)"            \
     --define=IMPL_VERSION=$(VERSION_NUMBER) \
     --define=SPEC_VERSION=$(VERSION_NUMBER) \
     $(if $3,$3,$(MANIFEST_TEMPLATE))        \
     > $(TMP_MANIFEST)
  $(JAR) -ufm $1 $(TMP_MANIFEST)
  $(RM) $(dir $(TMP_MANIFEST))
endef
```

Функция *add-manifest* оперирует файлом манифеста методом, подобным описанному выше. Сначала функция создаёт временный каталог, затем производит подстановку переменных в шаблоне файла манифеста. Затем функция обновляет архив и удаляет временный каталог. Обратите внимание на то, что последний аргумент функции является необязательным. Если путь к файлу манифеста не указан, функция использует значение переменной `MANIFEST_TEMPLATE`.

В универсальном *makefile*'е эти операции привязаны к общей функции, осуществляющей составление явного правила для создания Java-архива:

```
# $(call make-jar,jar-variable-prefix)
define make-jar
  .PHONY: $1 $$($1_name)
  $1: $($1_name)
  $$($1_name):
      cd $(OUTPUT_DIR); \
      $(JAR) $(JARFLAGS) $$(notdir $$@) $$($1_packages)
      $$(call add-manifest, $$@, $$($1_name), $$($1_manifest))
endef
```

Эта функция принимает один аргумент, префикс переменной *make*, который идентифицирует набор переменных, описывающих четыре параметра архива: имя цели, имя архива, пакеты архива и файл манифеста. Например, для создания архива *ui.jar* мы напишем следующее:

```
ui_jar_name     := $(OUTPUT_DIR)/lib/ui.jar
ui_jar_manifest := src/com/company/ui/manifest.mf
ui_jar_packages := src/com/company/ui \
                   src/com/company/lib

$(eval $(call make-jar,ui_jar))
```

Используя композицию имён переменных, мы можем сократить последовательность действий, выполняемых функцией, достигнув в тоже время гибкой её реализации.

Если нам нужно создать много архивов, мы можем автоматизировать этот процесс, поместив список имён архивов в переменную:

```
jar_list := server_jar ui_jar

.PHONY: jars $(jar_list)
jars: $(jar_list)

$(foreach j, $(jar_list),\
  $(eval $(call make-jar,$j)))
```

В некоторых случаях нам может понадобиться распаковать содержимое архива во временный каталог. Ниже представлен пример простой функции, реализующей это требование:

```
# $(call burst-jar, jar-file, target-directory)
define burst-jar
  $(call make-dir,$2)
  cd $2; $(JAR) -xf $1
endef
```

9.5 Справочные деревья и архивы сторонних разработчиков

Для того, чтобы использовать единое разделяемое справочное дерево с поддержкой создания разработчиками частичных рабочих копий, просто настройте механизм ночных сборок, создающий Java-архивы проекта, и включите эти архивы в `CLASSPATH` компилятора. После этого шага разработчики смогут сделать нужную им частичную рабочую копию и инициировать процесс компиляции (в предположении, что список исходных файлов создаётся динамически программой, подобной *find*). Когда компилятору Java нужно будет найти символ, определённый в отсутствующем исходном файле, компилятор произведёт поиск, основываясь на значении `CLASSPATH`, и обнаружит соответствующие файлы классов в архиве.

Получение Java-архивов сторонних разработчиков из справочного дерева реализуется также просто. Просто поместите пути к этим архивам в переменную `CLASSPATH`. Как уже было замечено, *makefile* может быть очень полезным инструментом управления этим процессом. Разумеется, функция *get-file* может быть использована для автоматического выбора стабильной или бета версии, локальных или удалённых Java-архивов при помощи соответствующего определения переменной `JAR_PATH`.

9.6 Enterprise JavaBeans

Java-компоненты уровня предприятия (Enterprise JavaBeans™, EJB) — это мощная техника инкапсуляции и повторного использования бизнес-логики, каркасом которой является механизм удалённых вызовов методов (Remote Method Invocation, RMI). EJB определяет Java-классы, используемые для реализации API сервера, используемого, в конечном счёте, удалёнными клиентами. Эти объекты и службы настраиваются при помощи специальных файлов в формате XML. После написания Java-класса и соответствующего ему конфигурационного XML-файла эти файлы нужно упаковать вместе в Java-архив. Затем вызывается специальный EJB-компилятор, создающий код заглушек и связок, реализующих поддержку RPC.

Следующий код может быть добавлен в код универсального *makefile*'а для предоставления поддержки EJB:

```
EJB_TMP_JAR = $(EJB_TMP_DIR)/temp.jar
META_INF    = $(EJB_TMP_DIR)/META-INF

# $(call compile-bean, jar-name,
#        bean-files-wildcard, manifest-name-opt)
define compile-bean
  $(eval EJB_TMP_DIR := $(shell mktemp -d \
                          $(TMPDIR)/compile-bean.XXXXXXXX))
  $(MKDIR) $(META_INF)
  $(if $(filter %.xml, $2),cp $(filter %.xml, $2) $(META_INF))
  cd $(OUTPUT_DIR) &&                      \
  $(JAR) -cf0 $(EJB_TMP_JAR)               \
         $(call jar-file-arg,$(META_INF)) \
         $(filter-out %.xml, $2)
  $(JVM) weblogic.ejbc $(EJB_TMP_JAR) $1
  $(call add-manifest,$(if $3,$3,$1),,)
  $(RM) $(EJB_TMP_DIR)
endef

# $(call jar-file-arg, jar-file)
jar-file-arg = -C "$(patsubst %/,%,$(dir $1))" $(notdir $1)
```

Функция *compile-bean* принимает три параметра: имя Java-архива, который требуется создать, список файлов, входящих в архив, и необязательный файл манифеста. Сначала при помощи программы *mktemp* создаётся пустой временный каталог, имя каталога сохраняется в переменной `EJB_TMP_DIR`. Поместив присваивание этой переменной в функцию *eval*, мы получаем гарантию того, что значение `EJB_TMP_DIR` будет указывать на новый временный каталог при каждом вычислении функции *compile-bean*. Поскольку функция *compile-bean* используется в командном сценарии, она будет вычисляться только при выполнении сценария. Затем функция осуществляет копирование всех XML файлов из списка `bean-files-`

`wildcard` в каталог *META-INF*. Именно в этом каталоге хранятся конфигурационные файлы EJB. После этого функция создаёт временный Java-архив, используемый в качестве входа для EJB-компилятора. Функция *jar-file-arg* преобразует имена вида *dir1/dir2/dir3* к виду *-C dir1/dir2 dir3*, поэтому относительные имена файлов архива корректны. Этот наиболее подходящий формат для передачи команде *jar* пути к каталогу *META-INF*. Поскольку XML-файлы, содержавшиеся в списке, уже скопированы в каталог *META-INF*, мы отсеиваем их из списка аргументов команды *jar* при помощи функции *filter-out*. После сборки временного архива вызывается EJB-компилятор WebLogic, создающий результирующий архив. Затем к составленному архиву добавляется файл манифеста. Последним действием является удаление временного архива.

Способ использования новой функции очевиден:

```
bean_files = com/company/bean/FooInterface.class      \
             com/company/bean/FooHome.class           \
             src/com/company/bean/ejb-jar.xml         \
             src/com/company/bean/weblogic-ejb-jar.xml

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
    $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

Список `bean_files` немного необычен. Пути к файлам классов, входящих в этот список, указаны относительно каталога *classes*, в то время как пути к XML-файлам будут вычисляться относительно каталога, в котором располагается *makefile*.

Это всё замечательно, но что если ваш архив содержит много файлов? Существует ли способ составить список файлов автоматически? Разумеется:

```
src_dirs := $(SOURCE_DIR)/com/company/...

bean_files =                                          \
  $(patsubst $(SOURCE_DIR)/%,%,                       \
    $(addsuffix /*.class,                             \
      $(sort                                          \
        $(dir                                         \
          $(wildcard                                  \
            $(addsuffix /*Home.java,$(src_dirs)))))))

.PHONY: ejb_jar $(EJB_JAR)
ejb_jar: $(EJB_JAR)
$(EJB_JAR):
    $(call compile-bean, $@, $(bean_files), weblogic.mf)
```

Этот код подразумевает, что список каталогов с исходными файлами хранится в переменной `src_dirs` (в списке могут находится и каталоги, не содержащие кода

EJB-компонентов), и что все файлы, оканчивающиеся строкой *Home.java*, идентифицируют пакеты, содержащие код EJB-компонентов. Выражение для определения переменной `bean_files` сначала добавляет суффикс шаблона к имени каждого каталога в списке, а затем вызывает функцию *wildcard* для нахождения всех файлов, имя которых оканчивается строкой *Home.java*. Имена файлов отбрасываются, полученный список каталогов сортируется, дублирующиеся элементы удаляются из списка. К каждому каталогу добавляется суффикс `/*.class`, в результате командный интерпретатор заменит шаблон списком соответствующих файлов классов. Наконец, от каждого элемента списка отсекается префикс, содержащий имя каталога с исходными файлами (поскольку такого подкаталога каталога *classes* не существует). Причиной использования шаблонов командного интерпретатора вместо функции *wildcard* является тот факт, что *make* не сможет гарантированно выполнить поиск файлов, соответствующих шаблону, *после* компиляции и генерации файлов классов. Если *make* вычислит функцию *wildcard* слишком рано, файлы не будут обнаружены, а кэш содержимого каталогов помешает найти эти файлы позже. Применение же функции *wildcard* в каталоге с исходными файлами совершенно безопасно, поскольку мы подразумеваем, что исходные файлы не будут добавляться во время работы *make*.

Предыдущий код будет работать в том случае, если у нас имеется небольшое число архивов компонентов. Другой стиль разработки подразумевает помещение каждого EJB-компонента в собственный Java-архив. Большие проекты могут содержать десятки архивов. Для того, чтобы осуществлять автоматическую обработку этой ситуации, нам нужно составить явное правило для каждого EJB-архива. В нашем примере исходный код EJB-компонентов самодостаточен: каждый компонент располагается в отдельном каталоге вместе с ассоциированным XML-файлом. Определить каталоги, содержащие EJB-компоненты, можно по наличию файлов, оканчивающихся строкой *Session.java*.

Основной подход заключается в поиске EJB-компонентов в каталогах с исходным кодом, построении явного правила для каждого компонента и записи этих правил в файл. Затем файл с правилами для EJB-компонентов включается в наш *makefile*. Создание файла с правилами для компонентов вызывается через механизм управления включаемыми файлами *make*.

```
# session_jars - архивы EJB, адресованные относительным путём.
session_jars =
  $(subst .java,.jar,                          \
    $(wildcard                                 \
      $(addsuffix /*Session.java, $(COMPILATION_DIRS))))

# EJBS - список всех EJB-архивов.
EJBS = $(addprefix $(TMP_DIR)/,$(notdir $(session_jars)))

# ejbs - Create all EJB jar files.
```

```
.PHONY: ejbs
ejbs: $(EJBS)
$(EJBS):
    $(call compile-bean,$@,$^,)
```

С помощью вызова функции *wildcard* со списком всех каталогов с исходным кодом в качестве аргумента мы находим все файлы, имя которых оканчивается на *Session.java*. В нашем примере имя архива образуется из имени найденного исходного файла с добавлением расширения *.jar*. Архивы будут помещаться во временный каталог. Переменная `EJBS` содержит список архивов, адресованных относительным путём от корня дерева бинарных файлов. Эти архивы являются целью, которую мы хотим обновить. Командным сценарием является вызов функции *compile-bean*, реализованной нами ранее. Фокус заключается в том, что список файлов указан в качестве реквизита каждого архива. Давайте посмотрим, как они будут создаваться.

```
-include $(OUTPUT_DIR)/ejb.d

# $(call ejb-rule, ejb-name)
ejb-rule = $(TMP_DIR)/$(notdir $1):            \
             $(addprefix $(OUTPUT_DIR)/,       \
               $(subst .java,.class,           \
                 $(wildcard $(dir $1)*.java))) \
             $(wildcard $(dir $1)*.xml)

# ejb.d - файл зависимостей EJB
$(OUTPUT_DIR)/ejb.d: Makefile
    @echo Вычисляю зависимости ejb...
    @for f in $(session_jars);       \
    do                               \
      echo "\$$(call ejb-rule,$$f)"; \
    done > $@
```

Зависимости для каждого EJB-архива записываются в файл *ejb.d*, включаемый в *makefile*. Когда *make* первый раз производит поиск этого файла, файл ещё не существует. Поэтому *make* вызывает правило для обновления включаемого файла. Это правило записывает по одной строке, подобной следующей, для каждого EJB-архива:

```
$(call ejb-rule,src/com/company/foo/FooSession.jar)
```

Результатом вычисления функции *ejb-rule* является имя целевого архива и списка реквизитов, как показано ниже:

```
classes/lib/FooSession.jar:                  \
    classes/com/company/foo/FooHome.jar      \
```

```
classes/com/company/foo/FooInterface.jar \
classes/com/company/foo/FooSession.jar   \
src/com/company/foo/ejb-jar.xml          \
src/com/company/foo/ejb-weblogic-jar.xml
```

Таким образом, *make* предоставляет возможность управлять довольно большим количеством архивов без необходимости ручной поддержки набора явных правил.

Глава 10

Повышаем производительность *make*

make имеет чрезвычайно важную роль в процессе разработки программного обеспечения. Компонуя составляющие проекта в приложение, *make* позволяет разработчикам избежать трудноуловимых ошибок, связанных со случайным пропуском какого-то шага сборки. Однако, если разработчики избегают использования *make* из-за низкой скорости выполнения сборки, все преимущества использования *make* теряются. Таким образом, чрезвычайно важно убедиться в том, что *makefile* был составлен с расчётом на максимальную производительность.

Проблемы производительности всегда довольно запутаны, однако, если взять в рассмотрение восприятие пользователей и различные пути выполнения кода, всё становится ещё сложнее. Не каждая цель в *make*-файле нуждается в оптимизации. В некоторых условиях даже радикальные оптимизации могут не оправдать затраченных на них усилий. Например, сокращение времени сборки с 90 до 45 минут может быть несущественным, поскольку даже с учётом оптимизации сборка становится операцией, начав которую, можно «идти на обед». С другой стороны, сокращение времени сборки с двух минут до одной может сопровождаться аплодисментами разработчиков, если во время сборки они вынуждены сидеть сложа руки.

При написании эффективных *make*-файлов важно знать стоимость различных операций, а также ясно понимать, какие именно из этих операций выполняются. В последующих разделах мы проведём несколько простых тестов производительности, позволяющих дополнить эти общие комментарии количественными данными и описать техники, помогающие найти узкие места (bottlenecks).

Другим подходом к повышению производительности является использование параллелизма и топологий локальных сетей. Одновременное исполнение более одного командного сценария (даже на одном процессоре) может существенно сокра-

тить время сборки.

10.1 Измеряем производительность

В этом разделе мы измерим производительность базовых операций *make*. Таблица 10.1 содержит результаты этих измерений. Далее будет рассмотрен каждый из тестов, а также представлены соображения по поводу влияния этих результатов на написанные вами *makefile*'ы.

Тесты для Windows запускались на Pentium 4 с тактовой частотой 1,9 ГГц (приблизительно 3578 BogoMips[1]) и оперативной памятью 512 Мб под управлением операционной системы Windows XP. Использовался Cygwin *make* версии 3.80, запускаемый из окна *rxvt*. Тесты для Linux запускались на Pentium 2 с тактовой частотой 450 ГГц (891 BogoMips) и оперативной памятью 256 Мб под управлением операционной системы Linux RedHat 9.

Командный интерпретатор, используемый *make*, может существенно повлиять на производительность выполнения *make*-файла. Командный интерпретатор *bash* сложен и обладает обширной функциональностью, поэтому он достаточно тяжело-

[1]Объяснение величины BogoMips можно найти на сайте `http://www.clifton.nl/bogomips.html` (прим. автора).

Операция	Повторений	Секунд на выполнение (Windows)	Число выполнений в секунду (Windows)	Секунд на выполнение (Linux)	Число выполнений в секунду (Linux)
make (bash)	1000	0,0436	22	0,0162	61
make (ash)	1000	0,0413	24	0,0151	66
make (bash)	1000	0,0452	22	0,0159	62
присваивание	10.000	0,0001	8130	0,0001	10.989
subst (short)	10.000	0,0003	3891	0,0003	3846
subst (long)	10.000	0,0018	547	0,0014	704
sed (bash)	1000	0,0910	10	0,0342	29
sed (ash)	1000	0,0699	14	0,0069	144
sed (sh)	1000	0,0911	10	0,0139	71
shell (bash)	1000	0,0398	25	0,0261	38
shell (ash)	1000	0,0253	39	0,0018	555
shell (sh)	1000	0,0399	25	0,0050	198

Таблица 10.1: Стоимость базовых операций

весен. Интерпретатор *ash* гораздо легче, он обладает меньшими возможностями, впрочем, вполне подходящими для большинства задач. Чтобы усложнить задачу, добавлю, что при запуске *bash* командой */bin/sh* его поведение существенно изменяется с целью максимального соответствия возможностям стандартного интерпретатора. На большинстве Linux-систем файл */bin/sh* является символической ссылкой на *bash*, в то время как в Cygwin эта ссылка указывает на *ash*. Для учёта этих различий некоторые тесты запускались трижды, по одному разу для каждого командного интерпретатора. Командный интерпретатор, использованный в тесте, указан в скобках. К примеру, «(sh)» означает, что использовался интерпретатор *bash*, запущенный при помощи символической ссылки */bin/sh*.

Первые три теста, обозначенные как «make», отображают стоимость запуска *make*, не совершающего полезной работы. *make*-файл содержит следующие строки:

```
SHELL := /bin/bash
.PHONY: x
x:
    $(MAKE) --no-print-directory --silent --question make-bash.mk; \
    ...Эта команда повторяется ещё 99 раз...
```

По необходимости слово «bash» заменялось соответствующим названием командного интерпретатора.

Мы использовали опции `--no-print-directory` и `--silent` для исключения ненужных вычислений, которые могли повлиять на время выполнения и затмить измеренные временные величины грудой бесполезного текста. Опция `--question` сообщает *make*, что выполнять команды не требуется, нужна только проверка зависимостей. В этом случае, если файл не требует обновления, *make* завершит работу с нулевым кодом возврата. Это позволяет *make* делать настолько мало работы, насколько это возможно. Этот *make*-файл не будет выполнять команд, зависимости в нём существуют только для одной абстрактной цели. Файл *make-bash.mk* выполняется родительским процессом *make* 10 раз. Содержимое этого файла представлено ниже:

```
define ten-times
  TESTS += $1
  .PHONY: $1
  $1:
      @echo $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2; \
```

```
      time $(MAKE) --no-print-directory --silent $2; \
      time $(MAKE) --no-print-directory --silent $2
endef

.PHONY: all
all:

$(eval $(call ten-times, make-bash, -f make-bash.mk))

all: $(TESTS)
```

После этого время, требуемое для тысячи запусков, усредняется.

Как вы можете видеть из таблицы, Cygwin *make* выполняется примерно 22 раза в секунду, или 0,044 секунд, в то время как под управлением операционной системы Linux (не смотря на гораздо более медленный процессор) выполнение осуществляется примерно 61 раз в секунду (т.е. одно выполнение занимает 0,016 секунд). Для проверки этих результатов был протестирован порт *make* под Windwos, не показавший, впрочем, существенного выигрыша в производительности. Заключение: хоть создание процесса Cygwin *make* и занимает немного больше времени, чем та же операция в Windows *make*, оба этих варианта значительно уступают в производительности аналогичной операции в Linux. Отсюда следует, что рекурсивное выполнение *make* под Windows может занимать значительно больше времени, чем рекурсивная сборка, запущенная под управлением Linux.

Как вы могли ожидать, используемый командный интерпретатор практически не влияет на скорость выполнения. Поскольку командный сценарий не содержит специальных символов, командный интерпретатор даже не вызывался. *make* выполнял команды самостоятельно. Это можно проверить, присвоив переменной `SHELL` произвольное значение и убедившись в том, что тест выполняется корректно. Разница в производительности при использовании различных интерпретаторов можно списать на нормальную вариацию времени выполнения процесса в системе.

Следующий тест измеряет время, требуемое для присваивания переменной значения — наиболее элементарной операции *make*. *make*-файл, называющийся *assign.mk*, содержит следующие строки:

```
# 10000 assignments
z := 10
...предыдущая строка повторяется 10000 раз...
.PHONY: x
x: ;
```

Этот *make*-файл выполняется в родительском *make*-файле с использованием нашей функции *ten-times*.

Очевидно, присваивание выполняется очень быстро. Cygwin *make* выполняет 8130 присваиваний в секунду, в то время как в системе Linux этот показатель

доходит до 10.989. Я уверен, что производительность выполнения этой операции в системе Windows на самом деле выше, чем показывают наши измерения, поскольку точное время создания десяти процессов *make* невозможно отделить от времени выполнения присваивания. Заключение: поскольку вероятность того, что в среднем *make*-файле будет осуществляться 10.000 присваиваний, довольно мала, стоимость выполнения присваиваний в среднем *make*-файле можно не учитывать.

Следующие два теста измеряют время выполнения функции *subst*. Первый тест осуществляет подстановку трёх символов в коротких строках, состоящих из десяти символов:

```
# 10000 подстановок в строке из 10 символов
dir := ab/cd/ef/g
x := $(subst /, ,$(dir))
...предыдущая строка повторяется 10000 раз...
.PHONY: x
x: ;
```

Операция занимает примерно в два раза больше времени чем простое присваивание, выполняясь в Windows 3891 раз в секунду. Повторюсь, показатели производительности в системе Linux значительно превосходят аналогичные показатели в Windows (как вы помните, производительность процессора компьютера, на котором установлена система Linux, примерно в четыре раза меньше производительности процессора компьютера с системой Windows).

Второй тест осуществляет примерно 100 подстановок в строке длиной в 1000 символов:

```
# Имя файла из 10 символов
dir := ab/cd/ef/g
# список путей из 1000 символов
p100 := $(dir);$(dir);$(dir);$(dir);$(dir);...
p1000 := $(p100)$(p100)$(p100)$(p100)$(p100)...

# 10000 подстановок в строке длиной в 1000 символов
x := $(subst ;, ,$(p1000))
...предыдущая строка повторяется 10000 раз...
.PHONY: x
x: ;
```

Следующие три теста измеряют скорость той же подстановки при использовании *sed*. Содержимое тестового файла представлено ниже:

```
# 100 sed using bash
SHELL := /bin/bash

.PHONY: sed-bash
sed-bash:
echo '$(p1000)' | sed 's/;/ /g' > /dev/null
...предыдущая строка повторяется 100 раз...
```

Как и раньше, *make*-файл выполняется с помощью функции *ten-times*. В системе Windows *sed* выполняется примерно в 50 раз медленнее, чем функция *subst*. В системе Linux *sed* работает в 24 раза медленнее.

Если учесть время, затраченное на запуск командного интерпретатора, становится очевидным, что использование командного интерпретатора *ash* в Windows даёт небольшую прибавку в скорости. При использовании *ash sed* всего лишь в 39 раз медленнее *subst*! В Linux влияние используемого командного интерпретатора на скорость выполнения прослеживается более чётко. При использовании *ash sed* всего в пять раз медленнее *subst*. Здесь же можно проследить эффект замены *bash* на *sh*. В среде Cygwin разница между *bash*, вызванного через */bin/bash*, и *bash*, вызванного через */bin/sh*, не прослеживается. В Linux же */bin/sh* выполняется значительно быстрее.

Последний тест измеряет затраты на выполнение команды в дочернем командном интерпретаторе, вызывая команду `make shell`. *make*-файл содержит следующие строки:

```
# 100 $(shell ) using bash
SHELL := /bin/bash
x := $(shell :)
...предыдущая строка повторяется 100 раз...
.PHONY: x
x: ;
```

Впрочем, результаты были вполне предсказуемы. Система Windows работает медленнее, чем Linux, командный интерпретатор *ash* работает быстрее, чем *bash*. Выигрыш от использования *ash* выражен в этом тесте более ярко и составляет примерно 50%. В системе Linux наибольшая производительность достигается при использовании *ash*, наименьшая — при использовании *bash* (вызванного из файла */bin/bash*).

Тесты производительности являются неиссякаемым источником задач, тем не менее, сделанные нами измерения могут помочь нам извлечь некоторую полезную информацию. Создавайте столько переменных, сколько считаете нужным, если, конечно, они помогают упростить структуру *make*-файла, поскольку их использование практически ничего не стоит. Встроенные функции более предпочтительны, чем запуск внешних программ, даже если структура вашего кода обязывает вас последовательно выполнять вызовы функций *make*. Избегайте использования рекурсивного *make* или избыточного порождения процессов в Windows. Если вы работаете в Linux и вам нужно создавать множество процессов, используйте *ash*.

Наконец, запомните, что для большинства *make*-файлов справедливо следующее утверждение: время выполнения *make*-файла практически полностью определяется временем выполнения внешних программ, а вовсе не нагрузкой *make* и не структурой *make*-файла. Как правило, сокращение числа запусков внешних программ сократит и время выполнения *make*-файла.

10.2 Определяем и устраняем узкие места

Излишние задержки выполнения *make*-файла могут появляться по одной из трёх причин: неудачный выбор структуры *make*-файла, неверный анализ зависимостей, и неправильное использование функций и переменных *make*. Эти проблемы могут маскироваться функциями *make*, подобными *shell*, которые вызывают команды, но не печатают их в терминал, что существенно затрудняет поиск источника задержек.

Анализ зависимостей — это палка о двух концах. С одной стороны, выполнение полного анализа зависимостей может вызвать существенные задержки. Без специальной поддержки компилятора, предоставляемой, к примеру, *gcc* и *jikes*, создание файла зависимостей требует запуска внешней программы, что практически удваивает время компиляции[2]. Преимуществом полного анализа зависимостей является возможность *make* осуществлять меньшее количество компиляций. К сожалению, разработчики часто не верят, что эта возможность себя оправдает, и пишут *make*-файлы с неполной информацией о зависимостях. Этот компромисс почти всегда превращается в проблему, заставляющую других разработчиков платить за эту скупость вдвойне, компилируя больше кода, чем потребовалось бы, будь у *make* полная информация о зависимостях.

Чтобы сформулировать стратегию анализа зависимостей, начните с понимания зависимостей, присущих вашему проекту. Когда все зависимости осознаны, можно приступать к представлению этих зависимостей в *make*-файле (вычисленных или перечисленных вручную) и выбору сокращённых путей осуществления сборки. Хоть и не все представленные шаги являются легко осуществимыми, этот метод сам по себе является наиболее простым.

Когда вы определили структуру *make*-файла и необходимые зависимости, эффективность *make*-файла достигается за счёт обхода некоторых известных ловушек.

10.2.1 Выбор переменных: простые или рекурсивные

Одной из наиболее общих проблем, относящихся к производительности, является использование рекурсивных переменных. Например, поскольку код, приведённый ниже, использует оператор `=` вместо оператора `:=`, при каждом обращении к переменной `DATE` её значение будет вычисляться заново:

```
DATE = $(shell date +%F)
```

[2]На практике время компиляции линейно зависит от размера входного текста и практически всегда определяется скоростью операций ввода/вывода. Точно так же время вычисления зависимостей с помощью опции `-M` линейно зависит от размера файла и ограничено скоростью операций ввода/вывода.

Опция `+%F` сообщает программе *date*, что дату требуется возвращать в формате «гггг-мм-дд», таким образом, большинство пользователей не заметят эффекта от многократного вызова *date*. Разумеется, разработчики, засидевшиеся в офисе до полуночи, могут быть приятно удивлены.

Поскольку *make* не выводит команды, выполняемые при помощи функции *shell*, идентифицировать, что же именно выполняется, может быть довольно трудно. Определив переменную `SHELL` как `/bin/sh -x`, вы можете выявить все команды, выполняемые *make*.

Следующий *make*-файл создаёт каталог перед осуществлением прочих действий. Имя каталога составляется из слова «out» и текущей даты:

```
DATE = $(shell date +%F)
OUTPUT_DIR = out-$(DATE)
make-directories := \
    $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p $(OUTPUT_DIR))
all: ;
```

После запуска *make*-файла с отладочной опцией интерпретатора мы увидим следующий вывод:

```
$ make SHELL='/bin/sh -x'
+ date +%F
+ date +%F
+ '[' -d out-2004-03-30 ']'
+ mkdir -p out-2004-03-30
make: all is up to date.
```

Теперь отчётливо видно, что команда *date* выполняется дважды. Если вам часто требуется осуществлять подобного рода отладку, вы можете упростить её, используя конструкцию следующего вида:

```
ifdef DEBUG_SHELL
  SHELL = /bin/sh -x
endif
```

10.2.2 Отключаем @

Ещё одним способом сокрытия команд является модификатор `@`. Иногда бывает полезным отключить эту возможность. Это легко осуществить с помощью определения вспомогательной переменной `QUIET`, содержащей символ `@`, и использования этой переменной в командах:

```
ifndef VERBOSE
  QUIET := @
endif
...
target:
    $(QUIET) echo Собираю цель target...
```

Когда нужно будет увидеть команды, скрытые при помощи модификатора, просто определите переменную VERBOSE через интерфейс командной строки:

```
$ make VERBOSE=1
echo Собираю цель target...
Собираю цель target...
```

10.2.3 Ленивая инициализация

При использовании простых переменных в сочетании с функцией *shell*, *make* осуществляет вызовы функции *shell* во время чтения *make*-файла. Если таких вызовов много, или если они осуществляют сложные вычисления, выполнение *make* может существенно замедлиться. Время отклика *make* можно измерить, вызвав *make* со спецификацией несуществующей цели:

```
$ time make no-such-target
make: *** No rule to make target no-such-target. Stop.
real    0m0.058s
user    0m0.062s
sys     0m0.015s
```

Приведённый выше код измеряет время, добавляемое *make* к каждой выполняемой команде, даже если эта команда тривиальна или ошибочна.

Поскольку рекурсивные переменные вычисляются заново при каждом обращении к ним, существует тенденция оформлять результаты сложных вычислений в виде простых переменных. С другой стороны, такой подход увеличивает время отклика *make* при сборке любой цели. Похоже, существует необходимость в дополнительном виде переменных, правая часть которых вычисляется в точности один раз при первом обращении к переменной.

Пример, иллюстрирующий необходимость подобного рода инициализации, был приведён в функции *find-compilation-dir* в разделе «Быстрый подход: компиляция всех исходных файлов за один раз» главы 9:

```
# $(call find-compilation-dirs, root-directory)
find-compilation-dirs =                         \
  $(patsubst %/,%,                              \
    $(sort                                      \
      $(dir                                     \
        $(shell $(FIND) $1 -name '*.java'))))
PACKAGE_DIRS := $(call find-compilation-dirs, $(SOURCE_DIR))
```

В идеале нам хотелось бы осуществлять операцию find только один раз при первом обращении к переменной PACKAGE_DIR. Это можно назвать *ленивой инициализацией* (*lazy initialization*). Мы можем создать подобного рода переменную с помощью функции *eval*:

```
PACKAGE_DIRS = $(redefine-package-dirs) $(PACKAGE_DIRS)
redefine-package-dirs =                                  \
  $(eval PACKAGE_DIRS := $(call find-compilation-dirs, \
                           $(SOURCE_DIR)))
```

Этот подход заключается в определении `PACKAGE_DIR` как изначально рекурсивной переменной. При первом обращении к переменной вычисляется ресурсоёмкая функция, в данном случае *find-compilation-dir*, и переменная переопределяется как простая. Наконец, значение переменной (теперь уже простой) возвращается как результат обращения к первоначально рекурсивной переменной.

Давайте рассмотрим этот пример более детально:

1. Когда *make* считывает эти переменные, он просто сохраняет правые части их определений, так как обе переменные являются рекурсивными.

2. При первом обращении к переменной `PACKAGE_DIRS` *make* извлекает соответствующую правую часть и производит вычисление переменной `redefine-package-dirs`.

3. Значением переменной `redefine-package-dirs` является единственный вызов функции *eval*.

4. Тело функции *eval* переопределяет переменную `PACKAGE_DIRS` как простую переменную, присваивая ей результат вычисления функции *find-compilation-dirs*. Теперь `PACKAGE_DIRS` инициализирована списком каталогов.

5. Результатом вычисления функции *redefine-package-dir* является пустая строка (поскольку результатом вычисления функции *eval* также является пустая строка).

6. *make* продолжает вычислять изначальное значение переменной `PACKAGE_DIRS`. Остаётся только подставить значение переменной `PACKAGE_DIRS`. *make* производит поиск переменной, находит простую переменную и возвращает её значение.

Единственным по-настоящему тонким моментом в этом коде является предположение, согласно которому *make* вычисляет правую часть определения переменной слева направо. Если, к примеру, *make* решит вычислить выражение `$(PACKAGE_DIRS)` раньше выражения `$(redefine-package-dirs)`, этот код не будет работать.

Процедура, описанная мной выше, может быть преобразована в функцию *lazy-init*:

```
# $(call lazy-init,variable-name,value)
define lazy-init
```

```
  $1 = $$(redefine-$1) $$($1)
  redefine-$1 = $$(eval $1 := $2)
endef

# PACKAGE_DIRS - ленивое вычисление списка каталогов
$(eval                              \
  $(call lazy-init,PACKAGE_DIRS, \
    $$(call find-compilation-dirs,$(SOURCE_DIRS))))
```

10.3 Параллельное выполнение *make*

Ещё одним способом увеличения производительности сборок является использование параллелизма, присущего проблеме обработки *make*-файла. Большинство *make*-файлов предназначены для осуществления задач, многие из которых можно обрабатывать параллельно, например, компиляцию исходных файлов C в объектные или создание библиотек из объектных файлов. Более того, сама структура хорошо написанных *make*-файлов предоставляет всю необходимую информацию для автоматического управления конкурирующими процессами.

Следующий пример демонстрирует сборку нашей программы mp3 плеера с опцией управления задачами, `--jobs=2` (или `-j 2`). На рисунке 10.1 изображён тот же самый запуск *make*, представленный с помощью диаграммы UML. Использование опции `--jobs` сообщает *make*, что при возможности следует обновлять параллельно две цели. Когда *make* обновляет цели параллельно, он выводит команды в том порядке, в каком они выполняются, чередуя в выводе команды сборки разных целей. Это может затруднить чтение вывода *make*, осуществляющего параллельную сборку. Давайте внимательно рассмотрим вывод.

```
$ make -f ../ch07-separate-binaries/makefile --jobs=2

1  bison -y --defines ../ch07-separate-binaries/lib/db/playlist.y
2  flex -t ../ch07-separate-binaries/lib/db/scanner.l >
   lib/db/scanner.c
3  gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include
   -M ../ch07-separate-binaries/app/player/play_mp3.c | \
   sed 's,\(play_mp3\.o\) *:,app/player/\1 app/player/play_mp3.d: ,'
   > app/player/play_mp3.d.tmp
4  mv -f y.tab.c lib/db/playlist.c
5  mv -f y.tab.h lib/db/playlist.h
6  gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include
   -M ../ch07-separate-binaries/lib/codec/codec.c | \
   sed 's,\(codec\.o\) *:,lib/codec/\1 lib/codec/codec.d: ,' >
   lib/codec/codec.d.tmp
7  mv -f app/player/play_mp3.d.tmp app/player/play_mp3.d
8  gcc -I lib -I ../ch07-separate-binaries/lib
```

```
   -I ../ch07-separate-binaries/include -M lib/db/playlist.c | \
   sed 's,\(playlist\.o\) *:,lib/db/\1 lib/db/playlist.d: ,' >
   lib/db/playlist.d.tmp
9  mv -f lib/codec/codec.d.tmp lib/codec/codec.d
10 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include
   -M ../ch07-separate-binaries/lib/ui/ui.c | \
   sed 's,\(ui\.o\) *:,lib/ui/\1 lib/ui/ui.d: ,' > lib/ui/ui.d.tmp
11 mv -f lib/db/playlist.d.tmp lib/db/playlist.d
12 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include
   -M lib/db/scanner.c | \
   sed 's,\(scanner\.o\) *:,lib/db/\1 lib/db/scanner.d: ,' >
   lib/db/scanner.d.tmp
13 mv -f lib/ui/ui.d.tmp lib/ui/ui.d
14 mv -f lib/db/scanner.d.tmp lib/db/scanner.d
15 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include -c
   -o app/player/play_mp3.o
   ../ch07-separate-binaries/app/player/play_mp3.c
16 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include -c
   -o lib/codec/codec.o
   ../ch07-separate-binaries/lib/codec/codec.c
17 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include -c
   -o lib/db/playlist.o lib/db/playlist.c
18 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include -c
   -o lib/db/scanner.o lib/db/scanner.c
   ../ch07-separate-binaries/lib/db/scanner.l: In function yylex:
   ../ch07-separate-binaries/lib/db/scanner.l:9: warning:
   return makes integer from pointer without a cast
19 gcc -I lib -I ../ch07-separate-binaries/lib
   -I ../ch07-separate-binaries/include -c
   -o lib/ui/ui.o ../ch07-separate-binaries/lib/ui/ui.c
20 ar rv lib/codec/libcodec.a lib/codec/codec.o
   ar: creating lib/codec/libcodec.a
   a - lib/codec/codec.o
21 ar rv lib/db/libdb.a lib/db/playlist.o lib/db/scanner.o
   ar: creating lib/db/libdb.a
   a - lib/db/playlist.o
   a - lib/db/scanner.o
22 ar rv lib/ui/libui.a lib/ui/ui.o
   ar: creating lib/ui/libui.a
   a - lib/ui/ui.o
23 gcc app/player/play_mp3.o lib/codec/libcodec.a lib/db/libdb.a
   lib/ui/libui.a app/player/play_mp3
```

Сначала *make* должен сгенерировать исходные файлы и файлы зависимостей.

Рис. 10.1: Диаграмма выполнения *make* при `--jobs=2`

Два сгенерированных исходных файла являются выводом команд *yacc* и *lex* (команды 1 и 2 соответственно). Третья команда генерирует файл зависимостей для *play_mp3.c*, её выполнение начинается до завершения генерации файлов зависимостей для *playlist.c* или *scanner.c* (командами 4, 5, 8, 9, 12 и 14). Таким образом, *make* выполняет одновременно три задачи, не смотря на то, что опция командной строки требует одновременного выполнения двух задач.

Команды `mv` (4 и 5) завершают генерацию исходного файла *playlist.c*, начавшуюся первой командой. Команда 6 начинает генерацию очередного файла зависимостей. Каждый командный сценарий выполняется одним процессом *make*, однако каждая цель и каждый реквизит формируют отдельный поток. Таким образом, команда 7, являющаяся второй командой сценария генерации файла зависимостей, исполняется тем же процессом *make*, что и команда 3. Команда 6 выполняется, скорее всего, процессом *make*, порождённым сразу после выполнения команд 1-4-5 (обработки грамматики *yacc*), но до генерации файла зависимостей, осуществляющейся командой 8.

Определение зависимостей продолжается в том же духе вплоть до команды 14. Все файлы зависимостей должны быть созданы до того, как *make* сможет приступить к следующей фазе обработки — повторному считыванию *make*-файла. Эта фаза образует естественную точку синхронизации.

Как только завершается повторное чтение *make*-файла, *make* может снова продолжить параллельное выполнение сборки. В этот раз *make* решает произвести компиляцию всех объектных файлов до того, как начать упаковывать их в библиотечные архивы. Этот порядок является недетерминированным. В следующий раз *make* может собрать библиотеку *libcodec.a* до того, как будет скомпилирован файл *playlist.c*, поскольку библиотека не требует наличия никаких объектных файлов, кроме *codec.o*. Таким образом, этот пример демонстрирует один порядок выполнения из множества возможных.

Наконец, происходит компоновка программы. В нашем случае фаза компоновки также является естественной точкой синхронизации и всегда будет происходить в последнюю очередь. Если, однако, целью была не единственная программа, а множество программ или библиотек, последняя исполняемая инструкция может также быть другой.

Запуск множества задач на многопроцессорном компьютере может иметь смысл, однако запуск более одной задачи на процессор может быть также очень полезным. Причина кроется в латентности дискового ввода/вывода и наличии кэширования на большинстве систем. Например, если процесс, к примеру, *gcc*, ожидает поступ-

ления данных с диска, данные для других задач (таких как *mv* или *yacc*) могут располагаться в памяти компьютера. В этом случае лучшим выходом будет разрешить задаче обработку данных. В общем случае запуск *make* с несколькими потоками выполнения на однопроцессорной системе практически всегда быстрее, чем запуск однопоточной сборки, и не так уж редко запуск трёх или даже четырёх потоков приводит к лучшему результату, чем при запуске двух потоков.

Опция `--jobs` может использоваться без аргумента. В этом случае *make* порождает по одному потоку на каждую обновляемую цель. Как правило, это плохая идея, поскольку на переключение контекста может уходить настолько много времени, что итоговая производительность будет гораздо ниже, чем в случае однопоточного выполнения.

Ещё одним способом управления множеством задач является использование средней загрузки системы как отправной точки. Средняя загрузка системы — это число запущенных задач, усреднённое за некоторый промежуток времени (как правило,1, 5 или 15 минут). Средняя загрузка выражается как число с плавающей точкой. Опция `--load-average` (или `-l`) задаёт верхнюю границу числа порождаемых задач. Например, следующая команда:

```
$ make --load-average=3.5
```

сообщает *make*, что потоки задач должны порождаться таким образом, чтобы средняя средняя загрузка системы была не более 3.5. Если средняя загрузка выше, *make* будет ждать, пока она не уменьшится до допустимых пределов, или пока все потоки не закончат свою работу.

Когда вы пишите *make*-файл для параллельного выполнения, задача правильного указания реквизитов становятся ещё более важной. Как уже было замечено, когда опция `--jobs` имеет значение 1, список реквизитов обычно вычисляется слева направо. Когда `--jobs` больше 1, реквизиты могут вычисляться параллельно. Поэтому любые отношения зависимости, неявно присутствующие в порядке вычисления реквизитов, при параллельном запуске должны быть указаны явно.

Ещё одной неприятностью при использовании параллельных сборок является проблема разделяемых временных файлов. Например, если каталог содержит файлы *foo.y* и *bar.y*, параллельный запуск *yacc* может привести к тому, что один из экземпляров файла *y.tab.c* или *y.tab.h* перепишет другой. С подобной проблемой вы также сталкиваетесь при использовании в своих сценариях временных файлов с фиксированными именами.

Ещё одной идиомой, препятствующей параллельному выполнению, является рекурсивный вызов *make* из цикла `for`:

```
dir:
    for d in $(SUBDIRS);        \
    do                          \
```

```
        $(MAKE) --directory=$$d; \
    done
```

Как уже упоминалось в разделе Рекурсивный *make* главы 6, *make* не может выполнять рекурсивные вызовы параллельно. Чтобы достичь параллельного выполнения, объявите каталоги абстрактными целями:

```
.PHONY: $(SUBDIRS)
$(SUBDIRS):
    $(MAKE) --directory=$@
```

10.4 Распределённое выполнение *make*

GNU *make* поддерживает малоизвестную (и практически не тестированную) опцию для управления сборками, распределёнными среди нескольких рабочих станций, соединённых сетью. Этот функционал основан на библиотеке Customs, распространяемой с дистрибутивом *Pmake*. *Pmake* — это альтернативная версия *make*, реализованная Адамом де Буром (Adam de Boor) в 1989 году для операционной системы Sprite и всё ещё поддерживаемая Андреасом Столке (Andreas Stolcke). Библиотека Customs помогает распределить выполнение *make* между множеством компьютеров. GNU *make* включает поддержку этой библиотеки начиная с версии 3.77.

Чтобы включить поддержку библиотеки Customs, вам нужно собрать *make* из исходного кода. Инструкцию по осуществлению этого процесса можно найти в файле *README.customs* в дистрибутиве *make*. Сначала вам нужно загрузить дистрибутив *pmake* (URL указан в инструкции), затем собрать *make* с опцией `--with-customs`.

Сердцем библиотеки Customs является демон (daemon), запускаемый на каждом узле распределённой вычислительной сети *make*. Все узлы должны иметь доступ к разделяемой файловой системе, предоставляемый, например, NFS. Один экземпляр демона назначается управляющим. Управляющий процесс назначает задачи участникам вычислительной сети. Когда *make* запускается с опцией `--jobs` больше 1, *make* контактирует с управляющим процессом, вместе они порождают задачи, распределяя их среди доступных узлов сети.

Библиотека Customs поддерживает множество возможностей. Узлы могут группироваться по архитектуре и ранжироваться по производительности. Узлам могут назначаться произвольные атрибуты, и задачи могут назначаться на основании значений атрибутов и булевых операторов. В добавок к этому, такие характеристики работы узлов, как время простоя, свободное дисковое пространство, свободное пространство в разделе подкачки, текущая средняя загрузка могут быть посчитаны во время выполнения задач.

Если ваш проект реализован на C, C++ или Objective-C вам следует рассмотреть возможность применения программы *distcc* (*http://distcc.samba.org*), предназначенной для распределённой компиляции. *distcc* написана Мартином Пулом (Martin Pool) и другими программистами для ускорения сборок проекта Samba. Это законченное робастное решение для проектов, написанных на C, C++ или Objective-C. Для использования этого инструмента достаточно заменить компилятор C программой *distcc*:

```
$ make --jobs=8 CC=distcc
```

Для каждой компиляции *distcc* использует препроцессор для обработки исходного кода, затем отправляет результат другим узлам сети для компиляции. Наконец, удалённые узлы возвращают полученные объектные файлы управляющему процессу. Этот подход устраняет необходимость в разделяемой файловой системе, что существенно упрощает установку и конфигурацию.

Множество рабочих узлов или *добровольцев* можно указать несколькими способами. Наиболее простым является перечисление узлов-добровольцев в переменной окружения перед запуском *distcc*:

```
$ export DISTCC_HOSTS='localhost wasatch oops'
```

distcc имеет много опций для управления списком удалённых узлов, интеграцией с компилятором, управления компрессией, путями поиска, а также обнаружения и исправления ошибок.

Ещё одним инструментом увеличения скорости компиляции является программа *ccache*, написанная руководителем проекта Samba Эндрю Тридгеллом (Andrew Tridgell). Идея очень проста: *ccache* кэширует результаты предыдущих сборок. Перед осуществлением компиляции осуществляется проверка, содержит ли кэш нужные объектные файлы. Это не требует участие нескольких узлов сети, не требуется даже существование сети. Автор сообщает о 5-10 кратном ускорении основного процесса компиляции. Наиболее простым способом использования этого инструмента является переопределение команды компиляции через интерфейс командной строки:

```
$ make CC='ccache gcc'
```

ccache можно использовать совместно с *distcc* для ещё большего ускорения процесса сборки. В добавок ко всему, оба этих инструмента доступны в наборе инструментов Cygwin.

Глава 11

Примеры *make*-файлов

make-файлы, приведённые ранее в этой книге, достаточно хороши для промышленного использования и вполне могут буть адаптированы под более сложные требования. Тем не менее, стоит всё же рассмотреть *make*-файлы некоторых реальных проектов, чтобы оценить, что люди могут сделать с помощью *make* под давлением требований конкретных продуктов. В этой главе мы детально рассмотрим несколько *make*-файлов. Первый *makefile* использовался для сборки этой книги[1]. Второй — для сборки ядра Linux версии 2.6.7.

11.1 *makefile* этой книги

Написание книги о программировании само по себе является интересным упражнением в построении систем сборки. Текст книги состоит из множества файлов, каждый из которых требует соответствующей обработки. Примеры являются реальными программами, каждую из которых нужно запустить, получить из вывод, обработать его и включить в основной текст (благодаря этому вывод не нужно копировать и вставлять в текст, что избавляет от риска внесения ошибок). В процессе написания книги полезно иметь возможность просмотреть текст в различных форматах. Наконец, доставка рабочего материала требует архивирования. Разумеется, все шаги должны быть воспроизводимыми и относительно простыми в поддержке.

Похоже, это работа для *make*! Возможность применения для удивительно разнородных потребностей — одна из самых замечательных особенностей *make*. Эта книга была написана в формате DocBook (т.е. XML). *make* — это стандартный выбор при работе с TEX, LATEX и `troff`.

Следующий пример содержит полный *makefile* этой книги. В нём примерно 440 строк. *makefile* разделяется на следующие базовые задачи:

[1] Подразумевается оригинал книги, перевод был подготовлен при помощи других технологий *(прим. переводчика)*.

- Управление примерами.
- Предварительная обработка XML.
- Генерация документов в различных форматах.
- Проверка исходного кода.
- Базовые задачи поддержки.

```
# Сборка книги.
#
# Основные цели этого файла:
#
# show_pdf  Генерация pdf и запуск программы просмотра
# pdf       Генерация pdf
# print     Печать pdf
# show_html Генерация html и запуск программы просмотра
# html      Генерация html
# xml       Генерация xml
# release   Создание архива с релизом
# clean     Удаление файлов, созданных в процессе сборки
#

BOOK_DIR     := /test/book
SOURCE_DIR   := text
OUTPUT_DIR   := out
EXAMPLES_DIR := examples

QUIET = @

SHELL       =  shell
AWK         := awk
CP          := cp
EGREP       := egrep
HTML_VIEWER := cygstart
KILL        := /bin/kill
M4          := m4
MV          := mv
PDF_VIEWER  := cygstart
RM          := rm -f
MKDIR       := mkdir -p
LNDIR       := lndir
SED         := sed
SORT        := sort
TOUCH       := touch
XMLTO       := xmlto
XMLTO_FLAGS =  -o $(OUTPUT_DIR) $(XML_VERBOSE)
process-pgm := bin/process-include
make-depend := bin/make-depend
```

```
m4-macros := text/macros.m4

# $(call process-includes, input-file, output-file)
# Осуществляет замену символов табуляции пробелами,
# подстановку макросов и обработку директив включения.
define process-includes
  expand $1 |                                              \
  $(M4) --prefix-builtins --include=text $(m4-macros) - | \
  $(process-pgm) > $2
endef

# $(call file-exists, file-name)
# Возвращает ненулевое значение в случае существования
# файла с заданным именем
file-exists = $(wildcard $1)

# $(call maybe-mkdir, directory-name-opt)
# Создаёт каталог, если он ещё не существует.
# Если параметр directory-name-opt опущен, в качестве имени
# каталога используется значение $@.
maybe-mkdir = $(if $(call file-exists,        \
                    $(if $1,$1,$(dir $@))),, \
                $(MKDIR) $(if $1,$1,$(dir $@)))

# $(kill-acroread)
# Завершает процесс Acrobat Reader
define kill-acroread
  $(QUIET) ps -W |                                  \
  $(AWK) 'BEGIN { FIELDWIDTHS = "9 47 100" }       \
          /AcroRd32/ {                              \
                       print "Killing " $$3;       \
                       system( "$(KILL) -f " $$1 ) \
                     }'
endef

# $(call source-to-output, file-name)
# Преобразует имя исходного файла в имя выходного файла.
define source-to-output
$(subst $(SOURCE_DIR),$(OUTPUT_DIR),$1)
endef

# $(call run-script-example, script-name, output-file)
# Запускает makefile примера.
define run-script-example
  ( cd $(dir $1);                                    \
    $(notdir $1) 2>&1 |                              \
    if $(EGREP) --silent '\$$\(MAKE\)' [mM]akefile; \
    then                                             \
      $(SED) -e 's/^++*/$$/';                        \
```

```
    else                                            \
      $(SED) -e 's/^++*/$$/'                        \
             -e '/ing directory /d'                 \
             -e 's/\[[0-9]\]//';                    \
    fi )                                            \
  > $(TMP)/out.$$$$ &                               \
  $(MV) $(TMP)/out.$$$$ $2
endef

# $(call generic-program-example,example-directory)
# Создаёт общие правила сборки примера.
define generic-program-example
  $(eval $1_dir      := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, \
                          $(wildcard $(EXAMPLES_DIR)/$1/*)))
  $($1_run_out): $($1_make_out) $($1_run_run)
      $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
      $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
      $(RM) -r $($1_dir)
      $(MKDIR) $($1_dir)
      $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
      $(TOUCH) $$@

  $($1_run_make):
      printf "#! /bin/bash -x\nmake\n" > $$@
endef

# Конечные форматы книги
BOOK_XML_OUT     := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT    := $(subst xml,html,$(BOOK_XML_OUT))
BOOK_FO_OUT      := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT     := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC      := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT      := $(call source-to-output,$(ALL_XML_SRC))
DEPENDENCY_FILES := $(call source-to-output,\
                          $(subst .xml,.d,$(ALL_XML_SRC)))
# xml/html/pdf - Производит желаемые конечные форматы книги.
.PHONY: xml html pdf
xml:  $(OUTPUT_DIR)/validate
html: $(BOOK_HTML_OUT)
pdf:  $(BOOK_PDF_OUT)
```

```
# show_pdf - Формирует pdf документ и отображает его.
.PHONY: show_pdf show_html print
show_pdf: $(BOOK_PDF_OUT)
    $(kill-acroread)
    $(PDF_VIEWER) $(BOOK_PDF_OUT)

# show_html - Создаёт html файл и отображает его.
show_html: $(BOOK_HTML_OUT)
    $(HTML_VIEWER) $(BOOK_HTML_OUT)

# print - Печатает заданные страницы книги.
print: $(BOOK_FO_OUT)
    $(kill-acroread)
    java -Dstart=15 -Dend=15 $(FOP) $< -print > /dev/null

# $(BOOK_PDF_OUT) - Формирует pdf файл.
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_HTML_OUT) - Формирует html файл.
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_FO_OUT) - Формирует временный fo-файл.
.INTERMEDIATE: $(BOOK_FO_OUT)
    $(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile

# $(BOOK_XML_OUT) - Обрабатывает все входные xml файлы.
$(BOOK_XML_OUT): Makefile

######################################################################
# Поддержка FOP
#
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - Определите этот макрос для просмотра вывода fop.
ifndef DEBUG_FOP
  FOP_FLAGS := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'       \
                         -e '/relative-align/d'        \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - Compute the appropriate CLASSPATH for fop.
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                  \
              $(subst ; ,;,                                   \
                $(addprefix c:/usr/xslt-process-2.2/java/,    \
                  $(addsuffix .jar;,                          \
                    xalan                                     \
                    xercesImpl                                \
```

```
                  batik                                   \
                  fop                                     \
                  jimi-1.0                                \
                  avalon-framework-cvs-20020315))))

# %.pdf - Шаблонное правило создания pdf из fo.
%.pdf: %.fo
    $(kill-acroread)
    java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - Шаблонное правило для создания fo из xml.
PAPER_SIZE := letter
%.fo: %.xml
    XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
    $(XMLTO) $(XMLTO_FLAGS) fo $<

# %.html - Шаблонное правило для создания html из xml.
%.html: %.xml
    $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<

# fop_help - Отображение справки по использованию fop.
.PHONY: fop_help
fop_help:
    -java org.apache.fop.apps.Fop -help
    -java org.apache.fop.apps.Fop -print help

######################################################################
# release - Создаёт релиз книги
#
RELEASE_TAR   := mpwm-$(shell date +%F).tar.gz
RELEASE_FILES := README Makefile *.pdf bin examples out text
.PHONY: release
release: $(BOOK_PDF_OUT)
    ln -sf $(BOOK_PDF_OUT) .
    tar --create                \
        --gzip                  \
        --file=$(RELEASE_TAR)   \
        --exclude=CVS           \
        --exclude=semantic.cache \
        --exclude=*~            \
        $(RELEASE_FILES)
    ls -l $(RELEASE_TAR)

######################################################################
# Правила для примеров из первой главы.
#
# Все каталоги с примерами.
EXAMPLES :=
    ch01-bogus-tab
    ch01-cw1
```

```
    ch01-hello
    ch01-cw2
    ch01-cw2a
    ch02-cw3
    ch02-cw4
    ch02-cw4a
    ch02-cw5
    ch02-cw5a
    ch02-cw5b
    ch02-cw6
    ch02-make-clean
    ch03-assert-not-null
    ch03-debug-trace
    ch03-debug-trace-1
    ch03-debug-trace-2
    ch03-filter-failure
    ch03-find-program-1
    ch03-find-program-2
    ch03-findstring-1
    ch03-grep
    ch03-include
    ch03-invalid-variable
    ch03-kill-acroread
    ch03-kill-program
    ch03-letters
    ch03-program-variables-1
    ch03-program-variables-2
    ch03-program-variables-3
    ch03-program-variables-5
    ch03-scoping-issue
    ch03-shell
    ch03-trailing-space
    ch04-extent
    ch04-for-loop-1
    ch04-for-loop-2
    ch04-for-loop-3
    ch06-simple
    appb-defstruct
    appb-arithmetic

# Я бы с удовольствием использовал этот цикл foreach, но ошибка
# в версии 3.80 приводит к аварийному останову.
#$(foreach e,$(EXAMPLES),$(eval $(call generic-program-example,$e)))

# Вместо этого приходитя раскрывать цикл вручную:
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
$(eval $(call generic-program-example,ch01-cw2a))
```

```
$(eval $(call generic-program-example,ch02-cw3))
$(eval $(call generic-program-example,ch02-cw4))
$(eval $(call generic-program-example,ch02-cw4a))
$(eval $(call generic-program-example,ch02-cw5))
$(eval $(call generic-program-example,ch02-cw5a))
$(eval $(call generic-program-example,ch02-cw5b))
$(eval $(call generic-program-example,ch02-cw6))
$(eval $(call generic-program-example,ch02-make-clean))
$(eval $(call generic-program-example,ch03-assert-not-null))
$(eval $(call generic-program-example,ch03-debug-trace))
$(eval $(call generic-program-example,ch03-debug-trace-1))
$(eval $(call generic-program-example,ch03-debug-trace-2))
$(eval $(call generic-program-example,ch03-filter-failure))
$(eval $(call generic-program-example,ch03-find-program-1))
$(eval $(call generic-program-example,ch03-find-program-2))
$(eval $(call generic-program-example,ch03-findstring-1))
$(eval $(call generic-program-example,ch03-grep))
$(eval $(call generic-program-example,ch03-include))
$(eval $(call generic-program-example,ch03-invalid-variable))
$(eval $(call generic-program-example,ch03-kill-acroread))
$(eval $(call generic-program-example,ch03-kill-program))
$(eval $(call generic-program-example,ch03-letters))
$(eval $(call generic-program-example,ch03-program-variables-1))
$(eval $(call generic-program-example,ch03-program-variables-2))
$(eval $(call generic-program-example,ch03-program-variables-3))
$(eval $(call generic-program-example,ch03-program-variables-5))
$(eval $(call generic-program-example,ch03-scoping-issue))
$(eval $(call generic-program-example,ch03-shell))
$(eval $(call generic-program-example,ch03-trailing-space))
$(eval $(call generic-program-example,ch04-extent))
$(eval $(call generic-program-example,ch04-for-loop-1))
$(eval $(call generic-program-example,ch04-for-loop-2))
$(eval $(call generic-program-example,ch04-for-loop-3))
$(eval $(call generic-program-example,ch06-simple))
$(eval $(call generic-program-example,ch10-echo-bash))
$(eval $(call generic-program-example,appb-defstruct))
$(eval $(call generic-program-example,appb-arithmetic))

######################################################################
# validate
#
# Производит проверку
# a) неподставленных макросов m4;
# b) символов табуляций;
# c) комментариев FIXME;
# d) RM: мои ответы Энди;
# e) дубликатов макросов m4.
#
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs      \
                     $(OUTPUT_DIR)/chk_fixme            \
```

```
                    $(OUTPUT_DIR)/chk_duplicate_macros \
                    $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
    $(TOUCH) $@

$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
    # Ищем макросы и символы табуляции...
    $(QUIET)! $(EGREP) --ignore-case          \
                       --line-number          \
                       --regexp='\b(m4_|mp_)' \
                       --regexp='\011'
                       $^
    $(TOUCH) $@

$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
    # Ищем комментарии RM: и FIXME...
    $(QUIET)$(AWK)                                                  \
            '/FIXME/  { printf "%s:%s: %s\n", FILENAME, NR, $$0 } \
             /^ *RM:/ {                                             \
                        if ( $$0 !~ /RM: Done/ )                    \
                        printf "%s:%s: %s\n", FILENAME, NR, $$0    \
                      }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
    $(TOUCH) $@

$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
    # Ищем повторные определения макросов...
    $(QUIET)! $(EGREP) --only-matching              \
        "\`[^']+'," $< |                            \
    $(SORT) |                                       \
    uniq -c |                                       \
    $(AWK) '$$1 > 1 { printf "$<:0: %s\n", $$0 }' | \
    $(EGREP) "^"
    $(TOUCH) $@

ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
    $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }' \
            $(filter %.d,$^) |                                 \
    $(SORT) -u |                                               \
    comm -13 - $(filter-out %.d,$^)
    $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
$(ALL_EXAMPLES):
    # Ищем неиспользуемые примеры...
    $(QUIET) ls -p $(EXAMPLES_DIR) | \
```

```
    $(AWK) '/CVS/ { next }           \
            /\// { print substr($$0, 1, length - 1) }' > $@
#####################################################################
# clean
#
clean:
    $(kill-acroread)
    $(RM) -r $(OUTPUT_DIR)
    $(RM) $(SOURCE_DIR)/*~ $(SOURCE_DIR)/*.log semantic.cache
    $(RM) book.pdf

#####################################################################
# Управление зависимостями
#
# Если выполняется цель clean, не генерируем и не читаем
# включаемые файлы.
#
ifneq "$(MAKECMDGOALS)" "clean"
  -include $(DEPENDENCY_FILES)
endif

vpath %.xml $(SOURCE_DIR)
vpath %.tif $(SOURCE_DIR)
vpath %.eps $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
    $(call process-includes, $<, $@)

$(OUTPUT_DIR)/%.tif: %.tif
    $(CP) $< $@

$(OUTPUT_DIR)/%.eps: %.eps
    $(CP) $< $@

$(OUTPUT_DIR)/%.d: %.xml $(make-depend)
    $(make-depend) $< > $@

#####################################################################
# Создание каталогов для вывода
#
# Создаём каталоги для вывода по мере необходимости.
#
DOCBOOK_IMAGES := $(OUTPUT_DIR)/release/images
DRAFT_PNG      := /usr/share/docbook-xsl/images/draft.png

ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                          \
    $(shell                                                      \
      $(MKDIR) $(DOCBOOK_IMAGES) &                               \
      $(CP) $(DRAFT_PNG) $(DOCBOOK_IMAGES);                      \
```

```
      if ! [[ $(foreach d,                                              \
                $(notdir                                                \
                  $(wildcard $(EXAMPLES_DIR)/ch*)),                     \
                -e $(OUTPUT_DIR)/$d &) -e . ]];                         \
      then                                                              \
        echo Компоновка примеров... > /dev/stderr;                 \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

Этот *makefile* написан для запуска в Cygwin без серьёзных претензий на переносимость в Unix. Тем не менее, я уверен, что в нём очень мало несовместимостей с Unix (если вообще есть), которые нельзя было бы решить переопределением значений переменных или, возможно, введением новых переменных.

Раздел глобальных переменных определяет расположение корневого каталога и относительные пути к каталогам с текстом книги, примерами и каталогу для вывода. Имя каждой нетривиальной программы, используемой в *make*-файле, определяется соответствующей переменной.

11.1.1 Управление примерами

Первая задача (управление примерами) является самой сложной. Каждый пример располагается в собственном подкаталоге каталога *book/examples/chn-<title>*. Каждый пример содержит собственный *makefile*. Для обработки примера мы сначала создаём каталог, содержащий символические ссылки на деревья каталогов выходных файлов, и работаем в нём. Потому артифакты, созданные в процессе работы *make*, не попадают в дерево каталогов с исходным кодом. Более того, бóльшая часть примеров для корректной работы требует, чтобы текущим рабочим каталогом был каталог с их *make*-файлом. После создания символических ссылок на каталоги с исходным кодом мы выполняем сценарий командного интерпретатора, `run-make`, вызывающий *makefile* с правильными аргументами. Если в каталоге с исходным кодом нет такого сценария, мы выполняем стандартную версию сценария. Вывод сценария `run-make` сохраняется в файле *make.out*. Некоторые примеры порождают исполняемые файлы, которые также нужно выполнить. Эта работа выполняется сценарием `run-run`, его вывод сохраняется в файле *run.out*.

Создание каталога с символическими ссылками осуществляется следующим кодом, находящимся в конце *make*-файла:

```
ifneq "$(MAKECMDGOALS)" "clean"
  _CREATE_OUTPUT_DIR :=                                                 \
    $(shell                                                             \
      $(MKDIR) $(DOCBOOK_IMAGES) &                                      \
      $(CP) $(DRAFT_PNG) $(DOCBOOK_IMAGES);                             \
      if ! [[ $(foreach d,                                              \
```

```
              $(notdir                                                  \
                $(wildcard $(EXAMPLES_DIR)/ch*)),                       \
              -e $(OUTPUT_DIR)/$d &) -e . ]];                           \
      then                                                              \
        echo Компоновка примеров... > /dev/stderr;                 \
        $(LNDIR) $(BOOK_DIR)/$(EXAMPLES_DIR) $(BOOK_DIR)/$(OUTPUT_DIR); \
      fi)
endif
```

Этот код осуществляет одно присваивание простой переменной, обёрнутое в директиву условной обработки `ifneq`. Условная директива нужна для того, чтобы *make* не создавал структуру каталогов в случае запуска команды `make clean`. На самом деле, переменная является фиктивной: её значение никогда не используется. Тем не менее, функция *shell* справа от оператора присваивания выполняется в процессе чтения *make*-файла. Эта функция проверяет существование каталога каждого примера в дереве выходных файлов. В случае отсутствия какого-либо каталога вызывается команда *lndir*, обновляющая каталог с символическими ссылками.

Тест, выполняемый командой `if`, заслуживает более тщательного анализа. Он состоит из одной проверки `-e` (существует ли каталог?) для каталога каждого из примеров. Реальный код выглядит примерно следующим образом: для нахождения всех примеров используется функция *wildcard*, затем имена каталогов примеров отсекаются функцией *notdir*, после чего для каждого каталога создаётся текст `-e $(OUTPUT_DIR)/`*`каталог`*` &&`. Все эти элементы объединяются и подставляются в условие `bash [[...]]`. Наконец, результат проверки условия инвертируется. Одно дополнительное условие, `-e .`, включается в качестве граничного условия, чтобы позволить циклу *foreach* просто добавить `&&` к каждому выражению.

Этого достаточно для того, чтобы убедится в том, что новые каталоги всегда будут включены в процесс сборки при обнаружении.

Следующим шагом является создание правил, обновляющие два выходных файла, *make.out* и *run.out*. Это осуществляется для *.out* файлов всех примеров при помощи следущей функции:

```
# $(call generic-program-example,example-directory)
# Создаёт общие правила сборки примера.
define generic-program-example
  $(eval $1_dir      := $(OUTPUT_DIR)/$1)
  $(eval $1_make_out := $($1_dir)/make.out)
  $(eval $1_run_out  := $($1_dir)/run.out)
  $(eval $1_clean    := $($1_dir)/clean)
  $(eval $1_run_make := $($1_dir)/run-make)
  $(eval $1_run_run  := $($1_dir)/run-run)
  $(eval $1_sources  := $(filter-out %/CVS, \
                          $(wildcard $(EXAMPLES_DIR)/$1/*)))
  $($1_run_out): $($1_make_out) $($1_run_run)
```

```
      $$(call run-script-example, $($1_run_run), $$@)

  $($1_make_out): $($1_clean) $($1_run_make)
      $$(call run-script-example, $($1_run_make), $$@)

  $($1_clean): $($1_sources) Makefile
      $(RM) -r $($1_dir)
      $(MKDIR) $($1_dir)
      $(LNDIR) -silent ../../$(EXAMPLES_DIR)/$1 $($1_dir)
      $(TOUCH) $$@

  $($1_run_make):
      printf "#! /bin/bash -x\nmake\n" > $$@
endef
```

Эта функция должна быть вызвана единожды с именем каталога каждого примера в качестве аргумента:

```
$(eval $(call generic-program-example,ch01-bogus-tab))
$(eval $(call generic-program-example,ch01-cw1))
$(eval $(call generic-program-example,ch01-hello))
$(eval $(call generic-program-example,ch01-cw2))
```

11.1.2 Обработка XML

Рискуя выставить себя перед потомками в дурном свете, хочу сообщить, что я не очень люблю формат XML. Я нахожу его неуклюжим и многословным. Поэтому когда я узнал, что рукопись должна быть написана в DocBook, я начал поиск более традиционных инструментов, которые смогли бы упростить мою работу. Препроцессор *m4* и *awk* — два инструмента, которые мне очень помогли.

Есть две проблемы, связанные с DocBook и XML, с которыми *m4* отлично справляется: неудобство многословного синтаксиса XML и необходимость управления идентификаторами, используемыми в перекрёстных ссылках. К примеру, чтобы выделить слово в DocBook, вам нужно написать:

```
<emphasis>not</emphasis>
```

Используя *m4*, я написал простой макрос, позволяющий записать тоже самое следующим образом:

```
mp_em(not)
```

Да, так уже лучше. В добавок я ввёл множество символических стилей форматирования, таких как `mp_variable` и `mp_target`. Это позволило мне выбрать тривиальный формат для этих сущностей (к примеру, отсутсвие выделения) и изменять его позже по желанию редактора без необходимости осуществлять поиск и замену по всему документу.

Возможно, поклонники XML завалят меня письмами с описанием решения этой задачи средствами XML (с помощью XML-сущностей или чего-нибудь в этом роде). Однако не стоит забывать, что Unix нужен, чтобы решать текущие задачи теми инструментами, которые у тебя есть. Как любит говорить Ларри Уолл (Larry Wall), «Есть более одного способа сделать это»(«There is more then one way to do it»). Кроме того, я опасаюсь, что чрезмерное изучение XML заморочит мне голову.

Вторая задача для *m4* — управление XML-идентификаторами, используемыми в перекрёстных ссылках. Каждая глава, раздел, пример и таблица имеют свой идентификатор:

```
<sect1 id="MPWM-CH-7-SECT-1">
```

Ссылки на раздел должны использовать этот идентификатор. С точки зрения программирования эта проблема довольно ясна. Идентификаторы являются сложными константами, расбросанными по всему «коду». Более того, символы сами по себе не имеют значения. Я не имею понятия, о чём может идти речь в первом разделе седьмой главы. Используя *m4*, я могу избежать дублирования сложных идентификаторов, используя вместо них имеющие смысл имена:

```
<sect1 id="mp_se_makedepend">
```

Что более важно, при смене нумерации разделов или глав (что в процессе написания книги происходило много раз) идентификатор нужно будет поменять только в одном файле. Это преимущество наиболее ощутимо при смене нумерации разделов в главе. Если бы я не ввёл символические ссылки, подобная операция могла бы потребовать полудюжены операций поиска и замены по всем тексту книги.

Вот несколько примеров макросов *m4*[2]:

```
m4_define(‘mp_tag’,    ‘<$1>‘$2’</$1>’)
m4_define(‘mp_lit’,    ‘mp_tag(literal, ‘$1’)’)
m4_define(‘mp_cmd’,    ‘mp_tag(command,‘$1’)’)
m4_define(‘mp_target’, ‘mp_lit($1)’)
m4_define(‘mp_all’,    ‘mp_target(all)’)
m4_define(‘mp_bash’,   ‘mp_cmd(bash)’)
m4_define(‘mp_ch_examples’,     ‘MPWM-CH-11’)
m4_define(‘mp_se_book’,         ‘MPWM-CH-11.1’)
m4_define(‘mp_ex_book_makefile’,‘MPWM-CH-11-EX-1’)
```

Ещё одной задачей предварительной обработки была реализация возможности включения текста примеров. Этот текст требует замены символов табуляций пробелами (посколько конвертер DocBook O'Reilly не может обрабатывать символы

[2]Префикс `mp` является сокращением Managing Projects (название книги), слова macro processor (макро-процессор) или make pretty (буквально «сделай красивым»). Выберите наиболее понравившийся вам вариант *(прим. автора)*.

табуляции, а в *make*-файлах их полно), обёртки содержимого в `<![CDATA[...]]>` для экранирования специальных символов, и, наконец, отсечения лишних символов переноса строки в начале и конце текста примеров. Я смог решить эту задачу благодаря следующей небольшой программе на *awk*, которую я назвал *process-includes*:

```
#! /usr/bin/awk -f
function expand_cdata( dir )
{
  start_place = match( $1, "include-" )
  if ( start_place > 0 )
  {
    prefix = substr( $1, 1, start_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }
  end_place = match( $2, "(</(programlisting|screen)>.*)$", tag )

  if ( end_place > 0 )
  {
    file = dir substr( $2, 1, end_place - 1 )
  }
  else
  {
    print "Bogus include '" $0 "'" > "/dev/stderr"
  }

  command = "expand " file

  printf "%s>&33;&91;CDATA[", prefix
  tail = 0
  previous_line = ""
  while ( (command | getline line) > 0 )
  {
    if ( tail )
      print previous_line;

    tail = 1
    previous_line = line
  }

  printf "%s&93;&93;&62;%s\n", previous_line, tag[1]
  close( command )
}

/include-program/ {
  expand_cdata( "examples/" )
  next;
```

```
}

/include-output/ {
  expand_cdata( "out/" )
  next;
}

/<(programlisting|screen)> *$/ {
  # Find the current indentation.
  offset = match( $0, "<(programlisting|screen)>" )

  # Strip newline from tag.
  printf $0

  # Read the program...
  tail = 0
  previous_line = ""
  while ( (getline line) > 0 )
  {
    if ( line ~ "</(programlisting|screen)>" )
    {
      gsub( /^ */, "", line )
      break
    }
    if ( tail )
      print previous_line

    tail = 1
    previous_line = substr( line, offset + 1 )
  }

  printf "%s%s\n", previous_line, line

  next
}

{
  print
}
```

Мы копируем XML-файлы из дерева каталогов с исходными файлами в каталог с выходными файлами, заменяем символы табуляции пробелами, производим подстановку макросов и производим включение файлов примеров:

```
process-pgm := bin/process-include
m4-macros   := text/macros.m4

# $(call process-includes, input-file, output-file)
# Осуществляет замену символов табуляции пробелами,
# подстановку макросов и обработку директив включения.
```

```
define process-includes
  expand $1 |                                                  \
  $(M4) --prefix-builtins --include=text $(m4-macros) - | \
  $(process-pgm) > $2
endef

vpath %.xml $(SOURCE_DIR)

$(OUTPUT_DIR)/%.xml: %.xml $(process-pgm) $(m4-macros)
    $(call process-includes, $<, $@)
```

Шаблонное правило определяет способ составления выходного XML-файла из исходного XML-файла. Оно также декларирует, что все выходные XML-файлы должны быть составлены заново, если макросы или файл сценария включения изменились.

11.1.3 Генерация документов

Пока весь код, который мы разобрали, не выполняет непосредственного форматирования текста и не производит документов, которые можно было бы напечатать или отобразить на экране. Очевидно, самой важной функцией *make*-файла является создание книги. Я заинтересован в двух выходных форматах: HTML и PDF.

Сначала мы рассмотрим, как происходит формарование HTML. Для этой задачи я использовал замечательную программу *xsltproc* и небольшой вспомогательный сценарий *xmlto*. Эти инструменты делают процесс трансформации достаточно простым:

```
# Выходные форматы книги
BOOK_XML_OUT  := $(OUTPUT_DIR)/book.xml
BOOK_HTML_OUT := $(subst xml,html,$(BOOK_XML_OUT))

ALL_XML_SRC   := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT   := $(call source-to-output,$(ALL_XML_SRC))

# html - Создаёт книгу в предпочтительном формате.
.PHONY: html
html: $(BOOK_HTML_OUT)

# show_html - Создаёт html-файл и отображает его.
.PHONY: show_html
show_html: $(BOOK_HTML_OUT)
    $(HTML_VIEWER) $(BOOK_HTML_OUT)

# $(BOOK_HTML_OUT) - Создаёт html file
$(BOOK_HTML_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile
```

```
# %.html - Шаблонное правило для создания html из xml.
%.html: %.xml
    $(XMLTO) $(XMLTO_FLAGS) html-nochunks $<
```

Шаблонное правило выполняет основную работу по конвертированию XML-файла в HTML-файл. Книга организована в виде одного главного файла, *book.xml*, который включает файлы каждой главы. Имя главного файла содержится в переменной `BOOK_XML_OUT`. HTML-аналогом является файл `BOOK_HTML_OUT`, являющийся целью в *make*-файле. XML-файлы являются реквизитами `BOOK_HMLT_OUT`. Для удобства определены две абстрактные цели: `html` и `show_html`. Первая цель создаёт HTML-файл, а вторая отображает его в браузере.

Несмотря на простоту идеи, создание PDF-файлов — значительно более сложная операция. Программа *xsltproc* может создавать PDF-файлы непосредственно, но у меня так и не получилось реализовать такой сценарий. Вся работа осуществлялась под управлением Cygwin в Windows, а Cygwin-версия *xsltproc* работает только с POSIX-путями. Специфическая версия DocBook, используемая мной, а также вся рукопись, использовали специфичные для Windows пути. Это отличие, насколько я понимаю, помешало *xsltproc* выполнить свою работу. Вместо этого я решил использовать *xsltproc* для генерации FO-XML файлов, которые Java-программа FOP (`http://xml.apache.org/fop`) может конвертировать в PDF.

Поэтому код генерации PDF немного длиннее:

```
# Выходные форматы
BOOK_XML_OUT := $(OUTPUT_DIR)/book.xml
BOOK_FO_OUT  := $(subst xml,fo,$(BOOK_XML_OUT))
BOOK_PDF_OUT := $(subst xml,pdf,$(BOOK_XML_OUT))
ALL_XML_SRC  := $(wildcard $(SOURCE_DIR)/*.xml)
ALL_XML_OUT  := $(call source-to-output,$(ALL_XML_SRC))

# pdf - Верстает книгу в предпочтительном формате.
.PHONY: pdf
pdf: $(BOOK_PDF_OUT)

# show_pdf - Формирует pdf документ и отображает его.
.PHONY: show_pdf
show_pdf: $(BOOK_PDF_OUT)
    $(kill-acroread)
    $(PDF_VIEWER) $(BOOK_PDF_OUT)

# $(BOOK_PDF_OUT) - Формирует pdf-файл.
$(BOOK_PDF_OUT): $(BOOK_FO_OUT) Makefile

# $(BOOK_FO_OUT) - Формирует промежуточный fo-файл.
.INTERMEDIATE: $(BOOK_FO_OUT)
$(BOOK_FO_OUT): $(ALL_XML_OUT) $(OUTPUT_DIR)/validate Makefile
```

```
#####################################################################
# Поддержка FOP
#
FOP := org.apache.fop.apps.Fop

# DEBUG_FOP - Определите этот макрос для просмотра вывода fop.
ifndef DEBUG_FOP
  FOP_FLAGS := -q
  FOP_OUTPUT := | $(SED) -e '/not implemented/d'       \
                         -e '/relative-align/d'        \
                         -e '/xsl-footnote-separator/d'
endif

# CLASSPATH - Compute the appropriate CLASSPATH for fop.
export CLASSPATH
CLASSPATH = $(patsubst %;,%,                                    \
              $(subst ; ,;,                                     \
                $(addprefix c:/usr/xslt-process-2.2/java/,  \
                  $(addsuffix .jar;,                            \
                    xalan                                       \
                    xercesImpl                                  \
                    batik                                       \
                    fop                                         \
                    jimi-1.0                                    \
                    avalon-framework-cvs-20020315))))

# %.pdf - Шаблонное правило создания pdf из fo.
%.pdf: %.fo
    $(kill-acroread)
    java -Xmx128M $(FOP) $(FOP_FLAGS) $< $@ $(FOP_OUTPUT)

# %.fo - Шаблонное правило для создания fo из xml.
PAPER_SIZE := letter
%.fo: %.xml
    XSLT_FLAGS="--stringparam paper.type $(PAPER_SIZE)" \
    $(XMLTO) $(XMLTO_FLAGS) fo $<

# fop_help - Отображение справки по использованию fop.
.PHONY: fop_help
fop_help:
    -java org.apache.fop.apps.Fop -help
    -java org.apache.fop.apps.Fop -print help
```

Как вы можете видеть, для отображения двух-фазового процесса сборки используется два шаблонных правила. Правило *.xml* → *.fo* вызывает *xmlto*, правило *.fo* → *.pdf* сначала закрывает все открытые окна приложения *Acrobat reader* (поскольку это приложение блокирует PDF-файлы, не позволяя приложению FOP перезаписать их), а затем запускает FOP. Поскольку программа FOP весьма «болтлива», а просмотр сотен строк бессмысленных предупреждений меня быстро уто-

мил, я добавил простой *sed*-фильтр, `FOP_OUTPUT`, чтобы удалить эти разражающие предупреждения. Тем не менее, иногда эти сообщения содержали что-то действительно полезное, поэтому я добавил отладочную переменную `DEBUG_FOP`, которая отключает мой фильтр. Наконец, как и в случае с HTML-версией, я включил пару удобных целей, `pdf` и `show_pdf`.

11.1.4 Проверка исходного кода

Учитывая аллергию DocBook на символы табуляции, инструкции макропроцессора и комментарии редакторов, проверка корректности исходного текста становится нетривиальной задачей. Чтобы упростить этот процесс, я реализовал четыре цели, задачей которых является проверка различных аспектов корректности документов.

```
validation_checks := $(OUTPUT_DIR)/chk_macros_tabs      \
                     $(OUTPUT_DIR)/chk_fixme            \
                     $(OUTPUT_DIR)/chk_duplicate_macros \
                     $(OUTPUT_DIR)/chk_orphaned_examples

.PHONY: validate-only
validate-only: $(OUTPUT_DIR)/validate
$(OUTPUT_DIR)/validate: $(validation_checks)
    $(TOUCH) $@
```

Каждая цель создаёт собственный файл с временной меткой, все такие файлы являются реквизитами основного файла *validate*.

```
$(OUTPUT_DIR)/chk_macros_tabs: $(ALL_XML_OUT)
    # Ищем макросы и символы табуляции...
    $(QUIET)! $(EGREP) --ignore-case          \
                       --line-number          \
                       --regexp='\b(m4_|mp_)' \
                       --regexp='\011'        \
                       $^
    $(TOUCH) $@
```

Первая проверка производит поиск макросов *m4*, подстановка которых по каким-то причинам не была осуществлена на этапе предварительной обработки. Это указывает либо на ошибку в имени макроса, либо на то, что макрос не был определён. Также производится поиск символов табуляции. Разумеется, ни одна их этих ситуаций не должна произойти, но прецеденты были. Одной из интересных деталей является использование восклицательного знака после `$(QUIET)`. Назначение этого восклицательного знака — инвертировать код возврата команды *egrep*. *make* должен считать команду ошибочной, если *egrep обнаружит* вхождение шаблона.

```
$(OUTPUT_DIR)/chk_fixme: $(ALL_XML_OUT)
    # Ищем комментарии RM: и FIXME...
    $(QUIET)$(AWK)
            '/FIXME/ { printf "%s:%s: %s\n", FILENAME, NR, $$0 } \
             /^ *RM:/ {                                          \
                      if ( $$0 !~ /RM: Done/ )                   \
                      printf "%s:%s: %s\n", FILENAME, NR, $$0    \
                    }' $(subst $(OUTPUT_DIR)/,$(SOURCE_DIR)/,$^)
    $(TOUCH) $@
```

Этот отрывок осуществляет поиск не исправленных мною заметок. Очевидно, любой текст, помеченный тегом `FIXME`, должен быть исправлен, а метка должна быть удалена. В добавок к этому, система должна оповещать при обнаружении вхождений метки `RM:`, за которой не следует сразу метка `Done`. Обратите внимание, что формат выдачи *printf* соответствует стандартному формату выдачи ошибок компиляции. Благодаря этому стандартные инструменты распознавания ошибок компиляции будут правильно обрабатывать эти предупреждения.

```
$(OUTPUT_DIR)/chk_duplicate_macros: $(SOURCE_DIR)/macros.m4
    # Ищем повторные определения макросов...
    $(QUIET)! $(EGREP) --only-matching             \
        "\[^]+'," $< |                             \
    $(SORT) |                                      \
    uniq -c |                                      \
    $(AWK) '$$1 > 1 { printf "$>:0: %s\n", $$0 }' | \
    $(EGREP) "^"
    $(TOUCH) $@
```

Этот отрывок осуществляет поиск повторных определений в файле *macros.m4*. Процессор макросов не считает повторное определение макроса ошибкой, поэтому я решил выполнять эту проверку самостоятельно. Проверка представляет из себя следующий конвеер: поиск объявлений макросов, сортировка имён по алфавиту, вычисление количества дубликатов, отсев строк, встречающихся только один раз, прогон через *egrep* исключительно ради кода возврата. Ещё раз обратите внимание на инвертирование кода возврата: *make* должен сообщить об ошибке только только в том случае, если дубликаты будут обнаружены.

```
ALL_EXAMPLES := $(TMP)/all_examples

$(OUTPUT_DIR)/chk_orphaned_examples: $(ALL_EXAMPLES) $(DEPENDENCY_FILES)
    $(QUIET)$(AWK) -F/ '/(EXAMPLES|OUTPUT)_DIR/ { print $$3 }' \
            $(filter %.d,$^) |                                 \
    $(SORT) -u |                                               \
    comm -13 - $(filter-out %.d,$^)
    $(TOUCH) $@

.INTERMEDIATE: $(ALL_EXAMPLES)
```

```
$(ALL_EXAMPLES):
    # Ищем неиспользованные примеры...
    $(QUIET) ls -p $(EXAMPLES_DIR) |  \
    $(AWK) '/CVS/ { next }            \
            /\// { print substr($$0, 1, length - 1) }' > $@
```

Последняя проверка осуществляет поиск примеров, на которые нет ссылок в тексте. Эта цель использует интересный приём. Она требует наличия двух наборов файлов: каталогов всех примеров и файлов зависимостей XML-файлов. Список реквизитов разбивается на два набора с помощью функций *filter* и *filter-out*. Список каталогов с примерами генерируется с помощью вызова `ls -p` (это команда добавляет слеш к именам каталогов) и поиска слешей в её выводе. Сначала осуществляется поиск файлов зависимостей в списке реквизитов, затем производится вывод найденных каталогов и удаление дубликатов. Это примеры, ссылки на которые есть в тексте. Полученный список отправляется на стандартных вход программе *comm*, а список всех известных каталогов с примерами помещается во второй файл. Опция `-13` означает, что *comm* должна печатать только строки из второй колонки (т.е. каталоги, на которые нет ссылок в файле зависимостей).

11.2 *makefile* ядра Linux

makefile ядра Linux является отличным примером использования *make* для сборки в рамках сложной инфраструктуры. Поскольку целью этой книги не является описание сруктуры и процесса сборки ядра Linux, мы рассмотим лишь несколько интересных применений *make* внутри системы сборки ядра. Более подробное обсуждением системы сборки ядра версий 2.5/2.6 и эволюции этой системы по сравнению с версией 2.4 вы можете найти по адресу `http://macarchive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Germaschewski-OLS2003.pdf`.

Поскольку упомянутый *makefile* имеет так много аспектов, мы обсудим лишь некоторые из тех, что могут быть использованы в различных приложениях. Сначала мы рассмотрим, как однобуквенные переменные *make* используются для симуляции ключей командной строки. Мы увидим, как разделить деревья каталогов с исходными и с бинарными файлами разделяются, чтобы пользователи смогли вызывать *make* прямо из каталога с исходным кодом. Затем мы исследуем методику, позволяющую *make*-файлу контролировать степень детализации вывода. Далее, мы рассмотрим наиболее интересные функции, определяемые пользователем, и увидим, как они препятствуют дублированию кода, улучшают читаемость кода и инкапсулируют сложность. Наконец, мы рассмотрим базовый функционал справки, реализованный с помощью *make*.

Ядро Linux следует известному шаблону *конфигурация, сборка, установка* (configure, build, install), применяемому большинством проектов свободного программного

обеспечения. В то время как множество открытых проектов используют отдельный сценарий *configure* (обычно созданный при помощи *autoconf*), ядро Linux реализует стадию конфигурации с помощью *make*, вызывая внешние сценарии и вспомогательные программы неявно.

Когда стадия конфигурации завершена, команда `make` или `make all` собирает ядро, все модули и создаёт сжатый образ ядра (цели `vmlinux`, `modules` и `bzImage`, соответственно). Каждой сборке ядра присваивается уникальный номер, хранящийся в файле *version.o*, прилинкованном к ядру. Это число (и файл *version.o*) обновляются средствами самого *make*-файла.

11.2.1 Опции командной строки

Первая часть *make*-файла содержит код, устанавливающий общие опции сборки, полученные из командной строки. Ниже приведена выдержка, осуществляющая контроль флага детализации вывода:

```
# Чтобы предупреждения были более заметными, по-умолчанию
# отображается минимум сообщений.
# Для более детального вывода используйте 'make V=1'.
ifdef V
  ifeq ("$(origin V)", "command line")
    KBUILD_VERBOSE = $(V)
  endif
endif
ifndef KBUILD_VERBOSE
  KBUILD_VERBOSE = 0
endif
```

Вложенная пара `ifdef`/`ifeq` проверяет, что переменная `KBUILD_VERBOSE` выставляется только в том случае, когда переменная `V` была определена в командной строке. Определение `V` в окружении или внутри *make*-файла не возымеет эффекта. Следующая директива `ifndef` выключает флаг `KBUILD_VERBOSE`, если его значение ещё не было определено. Чтобы включить детальный вывод из окружения или *make*-файла, вам нужно явно определить значение переменной `KBUILD_VERBOSE`, а не `V`.

Заметим, однако, что определение опции `KBUILD_VERBOSE` явно в командной строке разрешено и работает именно так, как вы ожидаете. Это может быть удобно для написания сценарев командного интерпретатора (или псевдонимов команд) для вызова *make*-файла. Использование полного имени будет более самодокументируемым и похожим на длинные опции GNU.

Другие опции командной строки, запрос запуска анализатора *sparse* (`C`) и требование сборки внешних модулей (`M`), используют аналогичную проверку, чтобы избежать случайного их переопределения внутри *make*-файла.

Следующая секция *make*-файла производит оработку опции, устанавливающей каталога вывода (O). Это довольно сложный участок кода. Чтобы прояснить его структуру, мы заменим некоторые части этого отрывка многоточиями:

```
# Система kbuild поддерживает сохранение выходных файлов в отдельном
# каталоге.
# Есть два пути воспользоваться этой возможностью. В обоих случаях
# рабочим каталогом должен быть каталог с исходным кодом ядра.
# 1) O=
# Используйте опцию O: "make O=dir/to/store/output/files/"
#
# 2) Определите  KBUILD_OUTPUT
# Определите переменную окружения KBUILD_OUTPUT так, чтобы она
# указывала на каталог, в который следует поместить выходные файлы.
# export KBUILD_OUTPUT=dir/to/store/output/files/
# make
#
# Опция O= имеет более высокий приоритет, чем переменная окружения
# KBUILD_OUTPUT.
# Переменная KBUILD_SRC выставляется в значение OBJ после старта make
# На данный момент не предполагается, что KBUILD_SRC будет
# использоваться пользователями без прав суперпользователя.
ifeq ($(KBUILD_SRC),)
  # Нас вызвали из каталога, в котором распологается исходный код
  # ядра. Требуется ли помещать объектные файлы в отдельный каталог?
  ifdef O
    ifeq ("$(origin O)", "command line")
      KBUILD_OUTPUT := $(O)
    endif
  endif
  ...
  ifneq ($(KBUILD_OUTPUT),)
    ...
    .PHONY: $(MAKECMDGOALS)
    $(filter-out _all,$(MAKECMDGOALS)) _all:
        $(if $(KBUILD_VERBOSE:1=),@)$(MAKE) -C $(KBUILD_OUTPUT)       \
        KBUILD_SRC=$(CURDIR)         KBUILD_VERBOSE=$(KBUILD_VERBOSE) \
        KBUILD_CHECK=$(KBUILD_CHECK) KBUILD_EXTMOD="$(KBUILD_EXTMOD)" \
        -f $(CURDIR)/Makefile $@
    # Поручаем сборку уже вызванному make
    skip-makefile := 1
  endif # ifneq ($(KBUILD_OUTPUT),)
endif # ifeq ($(KBUILD_SRC),)

# Оставшаяся часть makefile обрабатывается только в том случае, если
# текущий вызов make - последний.
ifeq ($(skip-makefile),)
  ...the rest of the makefile here...
endif
# skip-makefile
```

Вкратце, этот участок кода проверяет, определена ли переменная KBUILD_-OUTPUT. Если определена, в каталоге, имя которого содержится в KBUILD_OUT, вызывается *make* с определённой переменной KBUILD_SRC, содержащей путь к каталогу, в котором *make* был выполнен в первый раз. При этом для сборки используется исходный *makefile*. Также выставляется флаг skip-makefile, из-за чего оставшаяся часть *make*-файла не будет видна *make*. Рекурсивный *make* прочтёт тот же самый *makefile* ещё раз, только в этот раз переменная KBUILD_SRC будет определена, поэтому флаг skip-makefile не будет определён, и остаток *make*-файла будет прочитан и обработан.

На этом мы закончим рассмотрение опций командной строки. Большая часть *make*-файла находится в секции ifeq($(skip-makefile),).

11.2.2 Конфигурация или сборка?

makefile содержит цели для конфирурации и сборки. Конфигурационные цели имеют форму menuconfig, defconfig, и т.д. Вспомогательные цели, такие как clean, также трактуются как конфигурационные цели. Другие цели, такие как all, vmlinux и modules, являются целями сборки. Главным результатом вызова конфигурационных целей являются два файла: *.config* и *.config.cmd*. Эти два файла включаются *make*-файлом для целей сборки и не включаются для конфигурационных целей (поскольку именно конфигурационные цели создают эти файлы). Можно смешивать конфигурационные цели и цели сборки в одном вызове *make*:

```
\$ make oldconfig all
```

В этом случае *makefile* вызывает *make* рекурсивно для индивидуальной обработки каждой цели, таким образом обрабатывая конфигурационные цели отдельно от целей сборки.

Ниже приводится начало участка кода, управляющего конфигурацией, сборкой и смешением целей.

```
# Чтобы убедиться в том, что мы не включаем файл .config для
# конфигурационный целей, мы обрабатываем их заранее, передавая
# их scripts/kconfig/Makefile
# При вызове \GNUmake{} разрешается указывать несколько целей, а также
# смешивать конфигурационные цели и цели сборки.
# Пример: 'make oldconfig all'.
# Ситуацию со смешением целей нужно обрабатывать особым образом:
# производить повторный вызов make так, чтобы файл .config не
# включался для конфигурационных целей и в этом случае.

no-dot-config-targets := clean mrproper distclean \
                         cscope TAGS tags help %docs check%

config-targets := 0
```

```
mixed-targets  := 0
dot-config     := 1
```

Переменная `no-dot-config-targets` перечисляет дополнительные цели, не требующие наличие файла *.config*. Затем инициализируются переменные `config-targets`, `mixed-targets` и `dot-config`. Переменная `config-targets` равна единице, если в командной строке указаны цели, отвечающие за конфигурацию. Переменная `dot-config` равна единице, если в командной строке указаны цели, отвечающие за сборку ядра. Наконец, переменная `mixed-targets` равна 1, если были указаны оба типа целей.

Код определения `dot-config` выглядит следующим образом:

```
ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
  ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
    dot-config := 0
  endif
endif
```

Выражение *filter* будет непустым если значение `MAKECMDGOALS` содержит конфигурационные цели. Выражение `ifneq` будет истинным если результат вычисления *filter* будет непустым. Код довольно сложно читать, в частности, из-за наличия двойного отрицания. Выражение `ifeq` истинно, если значение `MAKECMDGOALS` содержит только конфигурационные цели. Таким образом, переменная `dot-config` будет выставлена в 0 если значение `MAKECMDGOALS` содержит только конфигурационные цели. Более многословная реализация может прояснить значение этих двух условий:

```
config-target-list := clean mrproper distclean \
                      cscope TAGS tags help %docs check%

config-target-goal := $(filter $(config-target-list), $(MAKECMDGOALS))
build-target-goal := $(filter-out $(config-target-list), $(MAKECMDGOALS))

ifdef config-target-goal
  ifndef build-target-goal
    dot-config := 0
  endif
endif
```

Использовать `ifdef` вместо *ifndef* допустимо, поскольку переменные с пустым значение трактуются как неопределённые, однако надо быть аккуратным и убедиться, что значение переменной не содержит пробелов (что сделает её определённой).

Переменные `config-targets` и `mixed-targets` выставляются в следующем блоке кода:

```
ifeq ($(KBUILD_EXTMOD),)
  ifneq ($(filter config %config,$(MAKECMDGOALS)),)
    config-targets := 1
    ifneq ($(filter-out config %config,$(MAKECMDGOALS)),)
      mixed-targets := 1
    endif
  endif
endif
```

Значение переменной `KBUILD_EXTMOD` будет непустым только если происходит сборка внешних модулей, во время обычных сборок она не будет определена. Условие в первой директиве `ifneq` будет истинным, если значение переменной `MAKECMDGOALS` содержит цель с суффиксом *config*. Условие во второй директиве `ifneq` будет истинным, если значение переменной `MAKECMDGOALS` содержит также и конфигурационные цели.

Определённые таким образом переменные используются в цепочке `if-else` в четырёх условных ветках. Код сокращен и отформатирован для выделения его структуры:

```
ifeq ($(mixed-targets),1)
  # make вызван со смешанным набором целей (конфигурационные и сборочные).
  # Обрабатываем их одну за другой.
  %:: FORCE
      $(Q)$(MAKE) -C $(srctree) KBUILD_SRC= $@
else
  ifeq ($(config-targets),1)
    # Указаны только конфигурационные цели. Убеждаемся, что реквизиты
    # обновлены, и запускаем рекурсивную сборку в scripts/kconfig для
    # сборки конфигурационных целей.
    %config: scripts_basic FORCE
        $(Q)$(MAKE) $(build)=scripts/kconfig $@
  else
    # Указаны только сборочные цели - это включает vmlinux, цели, зависящие
    # от архитектуры, clean и др. Это практически все цели, за исключением
    # целей с суффиксом config.
    ...
    ifeq ($(dot-config),1)
      # В этой секции нам потребуется .config
      # Считываем зависимости ко всем Kconfig-файлам, убеждаемся,
      # что oldconfig запущен при наличии изменений.
      -include .config.cmd
      include .config

      # Если .config требует пересборки, её нужно сделать через зависимость
      # autoconf от .config.
      # Чтобы избежать работы любых неявных правил, определим пустую комманду.
      .config: ;
```

```
      # Если .config модифицирован позднее include/linux/autoconf.h, кто-то
      # ковырялся в нём и забыл запустить make oldconfig.
      include/linux/autoconf.h: .config
          $(Q)$(MAKE) -f $(srctree)/Makefile silentoldconfig
    else
      # Цель без тела для использования в качестве реквизита
      include/linux/autoconf.h: ;
    endif
    include $(srctree)/arch/$(ARCH)/Makefile
    ... очень много make-кода ...
  endif #ifeq ($(config-targets),1)
endif #ifeq ($(mixed-targets),1)
```

Первая ветка, `ifeq ($(mixed-targets),1)`, обрабатывает случай смешанных целей в списке аргументов. Единственная цель в этой ветке — общее шаблонное правило. Поскольку специфические правила для обработки целей не определены (они все находятся в других ветках условных директив), каждая цель вызывает однократное срабатывание общего шаблонного правила. Таким образом, смешанный набор целей разделяется на несколько более простых, и для каждой цели рекурсивно вызывается *make*, который снова применяет ту же логику (только в этот раз командная строка не будет содержать смешанного набора целей). Вместо `.PHONY` используется реквизит `FORCE`, поскольку шаблонное правило вида

```
%:: FORCE
```

не может быть объявлено `.PHONY`. Поэтому выглядит логичным использовать `FORCE` вместо `.PHONY` во всём *make*-файле.

Вторая ветка цепочки `if-else`, `ifeq ($(config-targets),1)`, вызывается, если в командной строке присутствуют только конфигурационные цели. Единственным правилом в этой ветке является шаблонное правило `%command`. Соответствующая этому правилу команда рекурсивно вызывает *make* в подкаталоге *scripts/kconfig* и передаёт цель в качестве аргумента. Конструкция `$(build)` определена в конце *make*-файла:

```
# Сокращение для $(Q)$(MAKE) -f scripts/Makefile.build obj=dir
# Использование:
# $(Q)$(MAKE) $(build)=dir
build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/Makefile.build obj
```

Если переменная `KBUILD_SRC` определена, опция `-f` задаёт абсолютный путь к *make*-файлу, в противном случае используется относительный путь. При раскрытии макроса переменной `obj` будет присвоено значение справа от `$(build)=`.

В третьей ветке, `ifeq ($(dot-config),1)`, обрабатывает цели, требующие включения двух генерируемых конфигурационных файлов, *.config* и *.config.cmd*. Последняя ветка включает цель без тела `autoconf.h` для использования её в качестве реквизита, даже если соответсвтующий файл не существует.

Оставшаяся часть *make*-файла отвечает третьей и четвёртой условной ветке и содержит код сборки ядра и его модулей.

11.2.3 Управление командой `echo`

make-файлы ядра используют новаторскую технику управления уровнем детализации вывода, производимого выполняемыми комадами. Каждая важная задача представлена в двух вариантах: с тихим и с детальным режимом вывода. Делальная версия содержит только команду, которую нужно выполнить, в естественной форме и сохранена в переменной `cmd_`*`action`*. Тихая версия содержит короткое сообщение, описывающее выполняемое действие, и хранится в переменной `quiet_cmd_`*`action`*. Например, команада, создающая файл символов для *emacs*, выглядит следующим образом:

```
quiet_cmd_TAGS = MAKE $@
cmd_TAGS = $(all-sources) | etags -
```

Команда может быть выполнена с помощью вызова функции *cmd*:

```
# Если переменная quiet выставлена, использовать короткую версию команды
cmd = @$(if $($(quiet)cmd_$(1)),\
        echo ' $($(quiet)cmd_$(1))' &&) $(cmd_$(1))
```

Чтобы вызвать код, формирующий файл символов для *emacs*, *makefile* должен содержать следующий код:

```
TAGS:
    $(call cmd,TAGS)
```

Обратите внимание на то, что функция *cmd* начинается с символа `@`, поэтому единственный вывод, производимые ей, является выводом команды `echo`. В нормальном режиме переменная `queit` не определена, и условие `if, $($(quiet)cmd_$(1))` возвращает `$(cmd_TAGS)`. Поскольку эта переменная определена, результатом всего выражения будет команда

```
echo ' $(all-sources) | etags -' && $(all-sources) | etags -
```

Если тихий режим вывода более предпочтителен, переменная `queit` содержит текст `queit_`, и результатом вычисления функции будет выражение

```
echo ' MAKE $@' && $(all-sources) | etags -
```

Значением переменной также может быть `silent_`. Поскольку команда `silent_cmd_TAGS` не определена, вызов функции `cmd` ничего не выводит.

Вывод команд иногда более затруднителен, в частности, если команды содержат апострофы. В этом случае *makefile* содержит следующий код:

```
$(if $($(quiet)cmd_$(1)),echo ' $(subst ','\'',$($(quiet)cmd_$(1)))';)
```

Команда `echo` содержит подстановку, которая экранирует апострофы, благодаря чему они выводятся правильным образом.

Небольшие команды, не требующие определения переменных `cmd_` и `quiet_cmd_`, имеют префикс `$(0)`, который может быть пуст или равен `@`:

```
ifeq ($(KBUILD_VERBOSE),1)
  quiet =
  Q =
else
  quiet=quiet_
  Q = @
endif

# Если пользователь выполняет команду make -s ("тихий" режим),
# подавить вывод команд

ifneq ($(findstring s,$(MAKEFLAGS)),)
  quiet=silent_
endif
```

11.2.4 Функции, опредённые пользователем

makefile ядра определяет несколько функций. В этом разделе мы рассмотрим наиболее интересные из них. Форматирование кода было изменено для улучшения читабельности.

Функция *check_ gcc* используется для выбора опций командной строки *gcc*.

```
# $(call check_gcc,preferred-option,alternate-option)
check_gcc = \
  $(shell if $(CC) $(CFLAGS) $(1) -S -o /dev/null \
             -xc /dev/null > /dev/null 2>&1;      \
          then                                    \
            echo "$(1)";                          \
          else                                    \
            echo "$(2)";                          \
          fi ;)
```

Функция вызывает *gcc* с пустым списком исходных файлов с предопределёнными опциями командной строки. Выходной файл, а также содержимое стандартных потоков вывода и ошибки отбрасываются. Если выполнение *gcc* завершается успехом, это означает, что предопределённые опции командной строки допустимы на данной архитектуре, и функция возвращает эти опции в качестве результата. В противном случае опции являются недопустимыми, и функция возвращает альтернативный набор опций. Пример использования этой опции может быть найден в файле *arch/i386/Makefile*:

```
# Запрещаем gcc сохранять 16-байтовое выравнивание сегмента стека
CFLAGS += $(call check_gcc,-mpreferred-stack-boundary=2,)
```

Функция *if_ changed_ dep* генерирует информацию о зависимостях, используя довольно интересную технику.

```
# Выполняет команду и выполняет пост-обработку составленного
# .d файла зависимостей
if_changed_dep =                                        \
    $(if                                                \
      $(strip $?                                        \
        $(filter-out FORCE $(wildcard $^),$^)           \
        $(filter-out $(cmd_$(1)),$(cmd_$@))             \
        $(filter-out $(cmd_$@),$(cmd_$(1)))),           \
      @set -e;                                          \
      $(if $($(quiet)cmd_$(1)),                         \
        echo ' $(subst ','\'',$($(quiet)cmd_$(1)))';)   \
      $(cmd_$(1));                                      \
      scripts/basic/fixdep                              \
          $(depfile)                                    \
          $@                                            \
          '$(subst $$,$$$$,$(subst ','\'',$(cmd_$(1))))' \
          > $(@D)/.$(@F).tmp;                           \
      rm -f $(depfile);                                 \
      mv -f $(@D)/.$(@F).tmp $(@D)/.$(@F).cmd)
```

Функция состоит из одного условного выражения. Детали условия достаточно запутанны, однако, достаточно чётко выделяется намерение получить непустое значение, если файл с зависимостями должен быть обновлён. Обычно информация о зависимостях рассматривается в контексте даты последней модификации файлов. Система сборки ядра добавляет к этой задаче дополнительные ньюансы. Сборка ядра требует огромного количества опций компиляции для контроля сборки и поведения компонентов. Чтобы убедиться, что опции командной строки учитываются при сборке правильным образом, *makefile* производит перекомпиляцию файла при изменении опций для соответствующей цели. Перейдём к более детальному рассмотрению механизма, с помощью которого это реализовано.

Команда, используемая для компиляции каждого файла ядра, сохраняется в файле с расширением *.cmd*. Когда выполняется повторная сборка, *make* читает *.cmd*-файл и сравнивает текущую команду компиляции с предыдущей. Если они отличаются, в *.cmd*-файл записывается новое значение команды, что вызывает пересборку объектного файла. *.cmd* файл обычно состоит из двух частей: списка файлов-зависимостей целевого файла и одной переменной, содержащей список опций компилятора. Например, файл *arch/i386/cpu/mtrr/if.c* порождает следующий (сокращённый) *.cmd*-файл:

```
cmd_arch/i386/kernel/cpu/mtrr/if.o := gcc -Wp,-MD ...; if.c
```

```
deps_arch/i386/kernel/cpu/mtrr/if.o := \
arch/i386/kernel/cpu/mtrr/if.c \
...

arch/i386/kernel/cpu/mtrr/if.o: $(deps_arch/i386/kernel/cpu/mtrr/if.o)
$(deps_arch/i386/kernel/cpu/mtrr/if.o):
```

Вернёмся к функции *if_ changed_ dep*. Первый аргумент *strip* — это (возможно, пустой) список реквизитов, модифицированных позднее, чем цель. Второй аргумент — это все реквизиты, не являющиеся файлами или специальной целью `FORCE`. Предназначение последних двух вызовов *filter-out* определённо требует пояснения:

```
$(filter-out $(cmd\_$(1)),$(cmd\_$@))
$(filter-out $(cmd\_$@),$(cmd\_$(1)))
```

Результат вычисления этих вызовов будет непустой строкой, если аргументы командной строки изменились. Результатом подстановки макроса `$(cmd_$(1))` является текущая команда, а макроса `$(cmd_$@)` — предыдущая команда, например, переменная `cmd_arch/i386/kernel/cpu/mtrr/if.o` из предудущего примера. Если новая команда содержит дополнительные опции, результатом первого вызова *filter-out* будет пустая строка, а результатом второго — новые опции. Если же новая команда содержит меньше опций, первый результат будет содержать удалённые опции, а второй будет пустым. Заметим, что поскольку *filter-out* принимает список слов (каждое из которых интерпретируется как независимый шаблон), случай изменения порядка опций будет обработан корректно. Довольно изящное решение.

Первая инструкция в теле команды устанавливает опции интерпретатора, вызывающие немедленное завершение выполнения в случае ошибки. Это предотвращает повреждение файлов многострочными сценариями в случае возникновения проблем. В случае простых сценариев альтернативой может быть соединение инструкций оператором `&&`, а не точкой с запятой.

Следующая инструкция — это команда `echo`, записанная с использованием техники, описанной в разделе Управление командой `echo` текущей главы. Далее следует непосредственно команда генерации зависимостей, создающая файл `$(depfile)`, который затем трансформируется сценарием *scripts/basic/fixdep*. Вложенные в `fixdep` вызовы *subst* экранируют вхождения последовательности `$$` (командный интерпретатор сопоставляет ей идентификатор текущего процесса).

В случае отсутсвия ошибок вспомогательный файл `$(depfile)` удаляется, а сгенерированный файл зависимостей (с расширением *.cmd*) перемещается в соответствующий каталог.

Следующая функция, *if_ changed_ rule*, использует для управления командами ту же технику сравнения, что и *if_ changed_ dep*:

```
# Usage: $(call if_changed_rule,foo)
# will check if $(cmd_foo) changed, or any of the prequisites changed,
# and if so will execute $(rule_foo)
if_changed_rule =                                       \
    $(if $(strip $?                                     \
           $(filter-out $(cmd_$(1)),$(cmd_$(@F)))   \
           $(filter-out $(cmd_$(@F)),$(cmd_$(1)))), \
      @$(rule_$(1)))
```

Эта функция используется внутри макросов в *make*-файле верхнего уровня, чтобы слинковаять ядро:

```
# Нетривиальный момент: Если мы хотим произвести повторную
# линковку vmlinux, желательно увеличить номер версии, что
# означает перекомпиляцию init/version.o и линковку init/init.o.
# Однако, мы не можем сделать это во время фазы рекурсивной
# сборки (descending-into-subdirs phase), поскольку на этом
# этапе мы не можем знать, потребуется ли повторная линковка
# vmlinux. Поэтому мы снова рекурсивно запускаем make в каталоге
# init/ в рамках правила для vmlinux.

...

quiet_cmd_vmlinux__ = LD $@
define cmd_vmlinux__
  $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) \
  ...
endef

# set -e заставляет правило завершиться немедлненно
# в случае ошибки

define rule_vmlinux__
  +set -e;                                              \
  $(if $(filter .tmp_kallsyms%,$^),,                    \
    echo '  GEN    .version';                           \
    . $(srctree)/scripts/mkversion > .tmp_version;      \
    mv -f .tmp_version .version;                        \
    $(MAKE) $(build)=init;)                             \
  $(if $($(quiet)cmd_vmlinux__),                        \
    echo '  $($(quiet)cmd_vmlinux__)' &&)               \
  $(cmd_vmlinux__);                                     \
  echo 'cmd_$@ := $(cmd_vmlinux__)' > $(@D)/.$(@F).cmd
endef

define rule_vmlinux
  $(rule_vmlinux__);            \
  $(NM) $@ |                    \
  grep -v '\(compiled\)\|...' | \
  sort > System.map
```

```
endef
```

Функция *if_ changed_ rule* используется для вызова правила *rule_ vmlinux*, которое выполняет линковку и собирает финальный файл *System.map*. Как указано в комментариях к *make*-файлу, функция *rule_ vmlinux_ _* отвечает за генерацию версии ядра и повторную линковку *init.o* перед повторной линковкой *vmlinux* (за это отвечает первый оператор *if*). Второй опрератор *if* контролирует вывод команды линковки, `$(cmd_vmlinux__)`. Непосредственно выполняемая команда записывается в *.cmd*-файле для возможности сравнения при следующей сборке.

Глава 12

Отладка *make*-файлов

Отладка *make*-файлов стродни чёрной магии. К сожалению, не существует отладчика *make*-файлов, позволяющего проверить, как выполняется определённое провило, или как вычисляется значение переменной. Вместо этого основу отладки состовляет расстановка инструкций отладочной печати и «метод пристального взгляда». Тем не менее, GNU *make* предоставляет некоторую помощь в отладке в виде некоторых встроенных функций и опций командной строки.

Один из лучших способов отладки — добавление в код точек, в которые можно вклинить отладочные выводы, а также использование техник защищённого программирования (defensive programming techniques), позволяющих восстановить рабочее состояние сборки в случае возникновения непредвиденных обстоятельств. В этой главе я покажу несколько техник отладки и практик защищённого программирования, которые я нахожу наиболее полезными на практике.

12.1 Отладочные возможности *make*

Функция *warning* очень полезна для отладки *make*-файлов. Поскольку результатом вычисления этой функции является пустая строка, её можно помещать в любом месте *make*-файла: на самом верхнем уровне, в списке реквизитов или в секции команд правила. Это позволяет вам печатать значения переменных в том месте, где это наиболее удобно. Например:

```
$(warning Предупреждение верхнего уровня)

FOO := $(warning Правая часть простой переменной)bar
BAZ = $(warning Правая часть рекурсивной переменной)boo

$(warning Цель)target: $(warning Список реквизитов)makefile $(BAZ)
    $(warning Командный сценарий)
    ls
```

```
$(BAZ):
```

Производит следующий вывод:

```
$ make
makefile:1: Предупреждение верхнего уровня
makefile:2: Правая часть простой переменной
makefile:5: Цель
makefile:5: Список реквизитов
makefile:5: Правая часть рекурсивной переменной
makefile:8: Правая часть рекурсивной переменной
makefile:6: Командный сценарий
ls
makefile
```

Обратите внимание на вычисление функции *warning* следует обычной схеме аппликативных и отложенных вычислений *make*. Несмотря на то, что вычисление `BAZ` содержит вызов *warning*, соответствующее сообщение не будет напечатано до вычисление `BAZ` в списке реквизитов.

Возможность внедрять *warning* в любое место состовляет по сути основной отладочный механизм.

12.1.1 Опции командной строки

Следующие три опции командной строки я нахожу наиболее удобными для отладки: `--just-print (-n)`, `--print-data-base (-p)`, `--warn-undefined-variables`.

--just-print

Первым тестом, выполняемым мной для новой цели *make*-файла, это вызов *make* с опцией `--just-print (-n)`. Эта опция вынуждает *make* читать *makefile* и печатать все команды, которую он бы выполнял при сборке цели, без их непосредственного выполнения. Для удобства *make* также печатает все команды `echo`, помеченные модификатором подавления вывода `(@)`.

Эта опция подавляет выполнение всех команд. Однако вам нужно быть осторожным: хоть *make* и не будет выполнять сценарии правил, вызовы функции *shell* в аппликативном контексте всё же будут выполнены. Например:

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
             do                               \
               [[ -d $$d ]] || mkdir -p $$d;  \
             done)

$(objects) : $(sources)
```

Как мы уже видели раньше, назначение простой переменной `_MKDIRS` — создание необходимых каталогов. При выполнении с опцией `--just-print` *make* вызовет функцию *shell* во время чтения *make*-файла. Уже после этого *make* будет печатать (не выполняя) команды компиляции, необходимые для сборки файлов из списка `$(objects)`.

12.2 Отладочный код

12.3 Основные сообщения об ошибках

Предметный указатель