

Глава 13

Работа с файлами

В этой главе мы рассмотрим некоторые общие функции для манипулирования файлами. В стандартном релизе Эрланга есть множество функций для работы с файлами. Мы сосредоточимся на той малой их части, которую я использую в большинстве моих программ. Мы также рассмотрим некоторые примеры техники, которую я использую для написания эффективного кода обработки файлов. В дополнение к этому, я кратко упомяну некоторые реже используемые файловые операции, чтобы вы знали, что они есть. А если вы захотите узнать больше подробностей – обращайтесь к мануалам.

Мы сосредоточимся на следующих областях:

организация библиотек;

разные способы чтения файлов;

разные способы записи файлов;

работа с директориями;

поиск информации о файлах.

Организация библиотек

Функции для манипулирования файлами организованы в четыре модуля:

`file` – функции для открытия, закрытия, чтения и записи файлов; просмотра директорий и т. д. Краткая сводка часто используемых функций приведена в 2. За полными подробностями обращайтесь к руководству по модулю `file`.

`filename` – этот модуль содержит функции, которые манипулируют именами файлов платформонезависимым образом, так, что вы можете выполнять один и тот же код на разных операционных системах.

`filelib` – этот модуль – дополнение к `file`, который содержит ряд вспомогательных функций для просмотра файлов, проверки типов файлов и т. п. Большинство из них написаны, используя функции из `file`.

`io` – этот модуль содержит функции, которые работают с открытыми файлами. Он содержит функции для парсинга (разбора) данных в файле и записи форматированных

данных в файл.

change_group

Сменить группу у файла

change_owner

Сменить владельца у файла

change_time

Сменить время модификации или последнего доступа у файла

close

Закрыть файл

consult

Прочитать термы Эрланга из файла

copy

Копировать содержимое файла

del_dir

Удалить директорию

delete

Удалить файл

eval

Выполнить выражения Эрланга из файла

format_error

Вернуть строку с описанием причины ошибки

get_cwd

Получить текущую директорию

list_dir

Вывести список файлов в директории

make_dir

Создать директорию

make_link

Создать hard ссылку на файл

make_symlink

Создать soft (символическую) ссылку на файл

open

Открыть файл

position

Установить позицию в файле

pread

Читать из файла с указанной позиции

pwrite

Записать в файл с указанной позиции

read

Читать из файла

read_file

Прочитать весь файл целиком

read_file_info

Получить информацию о файле

read_link

Посмотреть — куда указывает ссылка

read_link_info

Получить информацию о ссылке или файле

rename

Переименовать файл

`script`

Выполнить и вернуть значение выражений Эрланга из файла

`set_cwd`

Установить текущую директорию

`sync`

Синхронизировать состояние файла в памяти и на физическом носителе

`truncate`

Обрезать файл

`write`

Записать в файл

`write_file`

Записать весь файл целиком

`write_file_info`

Сменить информацию о файле

Table 1: Сводка файловых операций (модуль `file`)

**

Разные способы чтения файлов

Давайте глянем — что у нас есть, когда дело доходит до чтения файлов. Мы начнём с написания пяти маленьких программ, которые открывают файл и берут оттуда данные несколькими разными способами.

Содержимое файла — это просто последовательность байт. Что они значат — зависит от интерпретации этих байтов.

Чтобы продемонстрировать это мы будем использовать один и тот же файл для всех наших примеров. Вообще-то, он содержит последовательность термов Эрланга. В зависимости от того, как мы открываем и читаем файл, мы можем интерпретировать содержимое как последовательность термов Эрланга, как последовательность

текстовых строк или как куски бессмысленных и беспощадных бинарных данных.

```
HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/data1.dat"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/data1.dat" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/data1.dat"dataHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/data1.dat"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/data1.dat"datHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/data1.dat":
```

```
{person, "joe" , "armstrong" ,
```

```
 [{occupation, programmer},
```

```
 {favoriteLanguage, erlang}}}.
```

```
{cat, {name, "zorro" },
```

```
 {owner, "joe" }}.
```

А теперь мы прочитаем части этого файла разными способами.

Чтение всех термов из файла

data1.dat содержит последовательность термов Эрланга. Мы можем прочитать их все, используя функцию `file:consult` следующим образом:

```
1> file:consult("data1.dat"). {ok,[{person,"joe", "armstrong", [{occupation,programmer},
{favoriteLanguage,erlang}]}, {cat,{name,"zorro"},{owner,"joe"}}]}
```

`file:consult(File)` полагает, что `File` содержит последовательность термов Эрланга. Она возвращает `{ok, [Term]}`, если может прочитать все термы из файла. В противном случае она возвращает `{error, Reason}`.

Чтение термов по одному за раз

Если мы хотим прочитать термы из файла по одному за раз, то мы открываем файл функцией `file:open`, а затем читаем отдельные термы функцией `io:read` до тех пор, пока не дойдём до конца файла. Затем мы закрываем файл функцией `file:close`.

Вот сеанс в оболочке Эрланга, который показывает что происходит, когда мы читаем термы из файла по одному за раз:

```
1> {ok, S} = file:open("data1.dat", read). {ok,<0.36.0>} 2> io:read(S, "). {ok,{person,"joe",
"armstrong", [{occupation,programmer},{favoriteLanguage,erlang}]}} 3> io:read(S, "). {ok,{cat,
{name,"zorro"},{owner,"joe"}}} 4> io:read(S, "). eof 5> file:close(S)
```

Функции, которые мы здесь использовали, следующие:

`@spec file:open(File, read) => {ok, IoDevice} | {error, Why}` Пытается открыть файл `File` для чтения. Возвращает `{ok, IoDevice}` в случае успеха, либо `{error, Reason}` в случае ошибки. `IoDevice` — это некий дескриптор, который используется для доступа к файлу.

`@spec io:read(IoDevice, Prompt) => {ok, Term} | {error, Why} | eof` Читает терм Эрланга из `IoDevice`. Подсказка `Prompt` игнорируется если `IoDevice` представляет собой открытый файл. Подсказка `Prompt` используется для выдачи подсказки только если мы используем `io:read` для чтения стандартного ввода.

`@spec file:close(IoDevice) => ok | {error, Why}`

Закрывает `IoDevice`.

Используя эти функции мы можем реализовать `file:consult`, который мы использовали в предыдущей части. Вот, как `file:consult` может быть определён:

HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl consult(File) -> case
file:open(File, read) of {ok, S} -> Val = consult1(S), file:close(S), {ok, Val}; {error, Why} ->
{error, Why} end.
```

```
consult1(S) -> case io:read(S, ") of {ok, Term} -> [Term|consult1(S)]; eof -> []; Error -> Error
end.
```

На самом деле `file:consult` определён не так. Стандартная библиотека использует улучшенную обработку ошибок.

Ну а теперь пришло время посмотреть на версию из стандартной библиотеки. Если вы поняли, как работает предыдущая версия функции, то вы легко поймёте код из библиотеки. Вот только есть одна проблема. Как найти исходный код для `file.erl`? Для нахождения кода мы используем функцию `code:which`, которая обнаруживает объектный код для любого загруженного модуля.

```
1> code:which(file). "/usr/local/lib/erlang/lib/kernel-2.11.2/ebin/file.beam"
```

В стандартном релизе у каждой библиотеки есть две поддиректории. Одна, называемая `src`, содержит исходный код. Другая, называемая `ebin`, содержит

скомпилированный Эрланг код. Так что исходный код для файла file.erl должен находиться в следующей директории:

```
/usr/local/lib/erlang/lib/kernel-2.11.2/src/file.erl
```

В случае, когда ничего уже не помогает, а документация не даёт ответов на ваши вопросы, быстрый взгляд в исходный код может помочь с ответом. Я знаю, что этого (поиска ответа в исходниках) не должно происходить, но все мы люди и иногда документация просто не помогает.

Чтение строк из файла по одной за раз

Если заменить `io:read` на `io:get_line`, то мы можем прочитать строки из файла по одной за раз. `io:get_line` читает символы до тех пор, пока не встретит символ перевода строки или конец файла. Вот пример:

```
1> {ok, S} = file:open("data1.dat", read). {ok,<0.43.0>} 2> io:get_line(S, "). "{person, \"joe\", \"armstrong\", \"n\" 3> io:get_line(S, "). \"\t[{occupation, programmer}, \"n\" 4> io:get_line(S, "). \"\t {favoriteLanguage, erlang}]]\". \"n\" 5> io:get_line(S, "). \"{cat, {name, \"zorro\"}}, \"n\" 7> io:get_line(S, "). \" {owner, \"joe\"}}\". \"n\" 8> io:get_line(S, "). eof 9> file:close(S). ok
```

Чтение всего файла целиком как бинарный объект

Вы можете использовать `file:read_file(File)`, чтобы прочитать файл целиком в бинарный объект, используя следующую атомарную операцию:

```
1> file:read_file("data1.dat"). {ok,<<\"{person, \"joe\", \"armstrong\"}...>>}
```

`file:read_file(File)` возвращает `{ok, Bin}` в случае успеха и `{error, Why}` в противном случае.

Это явно лучший способ чтения файлов и я использую этот способ наиболее часто. В большинстве случаев я читаю файл целиком в память одной операцией, работаю над содержимым файла и сохраняю файл тоже одной операцией (используя `file:write_file`). У нас будет пример для данного способа работы.

Чтение файла с произвольным доступом

Если файл, который мы хотим прочитать, очень большой или содержит бинарные данные в формате, который определён где-то ввне, то мы можем открыть файл в сыром (`raw`) режиме и читать порции файла операцией `file:pread`.

Пример:

```
1> {ok, S} = file:open("data1.dat", [read,binary,raw]). {ok,{file_descriptor,prim_file,
{#Port<0.106>,5}}}
```

```
2> file:pread(S, 22, 46). {ok,<<"rong\\",\\n\\t[{occupation, progr...>>} 3> file:pread(S, 1, 10). {ok,
<<"person, \\j">>} 4> file:pread(S, 2, 10). {ok,<<"erson, \\jo">>} 5> file:close(S).
```

file:pread(loDevice, Start, Len) читает точно Len байт из loDevice, начиная с байта в позиции Start (байты в файле нумеруются так, что первый байт находится в позиции 1) (прим. перев. - так в книге. В документации — всё по-другому). Она возвращает {ok, Bin} или {error, Why}.

В заключение, мы используем функции для произвольного доступа к файлу для написания утилиты, которая нам понадобится в следующей главе. В части 14.7 «Широковещательный сервер» мы разработаем простой широковещательный сервер (это сервер для так называемого потокового вещания. В данном случае — для вещания MP3). Часть этого сервера нуждается в поиске исполнителя и названий композиций, которые внедрены в файл MP3. Мы сделаем это в следующей части.

Чтение тегов MP3

MP3 — это бинарный формат, используемый для хранения сжатых звуковых данных. MP3 файлы сами по себе не содержат информацию о содержимом файла. К примеру в MP3 файле с музыкой не будет имени исполнителя. Эти данные (название композиции, исполнитель и прочее) хранятся внутри MP3 файла в специальном блочном формате ID3. Теги ID3 были придуманы программистом Eric Kemp для хранения метаданных, описывающих содержимое звукового файла. Вообще-то, есть несколько форматов ID3, но для наших целей мы будем использовать простейшие формы тегов — ID3v1 и ID3v1.1.

У тега ID3v1 простая структура — это последние 128 байт файла, содержащие тег фиксированной длины. Первые 3 байта содержат ASCII символы TAG, за которыми идут ряд полей фиксированной длины. Полностью эта 128 байтовая структура показана далее:

Длина

Содержимое

3

Заголовок, содержащий символы TAG

30

название

30

исполнитель

30

альбом

4

год

30

комментарий

1

жанр

В теге ID3v1 не было места, чтобы добавить номер композиции. Способ, реализующий это был предложен Michael Mutschler в формате ID3v1.1. Идея в том, чтобы заменить 30 байтовый комментарий следующим:

Длина

Содержимое

28

Комментарий

1

0 (ноль)

1

Номер композиции

Легко написать программу, которая будет читать теги ID3v1 из MP3 файла и сопоставлять поля, используя бинарное битовое сопоставление с образцом. Вот эта программа:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl"DownloadHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" id HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" 3_ HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" v HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" 1. HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/id3_v1.erl" erl -module(id3_v1). -import(lists,
[filter/2, map/2, reverse/1]). -export([test/0, dir/1, read_id3_tag/1]).
```

```
test() -> dir("/home/joe/music_keep").
```

```
dir(Dir) -> Files = lib_find:files(Dir, "*.mp3", true), L1 = map(fun(I) -> {I, (catch read_id3_tag(I))}
end, Files), %% L1 = [{File, Parse}] where Parse = error | [{Tag,Val}] %% we now have to
remove all the entries from L where %% Parse = error. We can do this with a filter operation
L2 = filter(fun({_,error}) -> false; () -> true end, L1), lib_misc:dump("mp3data", L2).
```

```
read_id3_tag(File) -> case file:open(File, [read,binary,raw]) of {ok, S} -> Size =
filelib:file_size(File), {ok, B2} = file:pread(S, Size-128, 128), Result = parse_v1_tag(B2),
file:close(S), Result; Error -> {File, Error} end.
```

```
parse_v1_tag(<<$T,$A,$G, Title:30/binary, Artist:30/binary, Album:30/binary, Year:4/binary,
Comment:28/binary, 0:8,Track:8,_Genre:8>>) ->
```

```
{"ID3v1.1", [{track,Track}, {title,trim(Title)}, {artist,trim(Artist)}, {album, trim(Album)}}};
parse_v1_tag(<<$T,$A,$G, Title:30/binary, Artist:30/binary, Album:30/binary, Year:4/binary,
Comment:30/binary,Genre:8>>) -> {"ID3v1", [{title,trim(Title)}, {artist,trim(Artist)}, {album,
trim(Album)}}}; parse_v1_tag() -> error.
```

```
trim(Bin) -> list_to_binary(trim_blanks(binary_to_list(Bin))).
```

```
trim_blanks(X) -> reverse(skip_blanks_and_zero(reverse(X))).
```

```
skip_blanks_and_zero([$s|T]) -> skip_blanks_and_zero(T); skip_blanks_and_zero([0|T]) ->
skip_blanks_and_zero(T); skip_blanks_and_zero(X) -> X.
```

Основная точка входа нашей программы — это `id3_v1:dir(Dir)`. Первое, что мы делаем — это ищем все наши MP3 файлы, вызывая `lib_find:find(Dir, "*.mp3", true)` (утилита поиска показана далее в части 13.8), которая рекурсивно сканирует директории ниже `Dir` на предмет файлов MP3. Найдя файл, мы разбираем теги, вызывая `read_id3_tag`. Разбор сильно упрощён, потому что мы используем простое битовое сопоставление с образцом. После этого мы подчищаем имена исполнителей и названия композиций, удаляя завершающие пробелы и нулевые символы, которые разделяют строки. В конце мы выводим результат в файл для дальнейшего использования (`lib_misc:dump` описывается в части E.2, Техника отладки).

Большинство музыкальных файлов помечены тегами ID3v1, даже если они

дополнительно содержат ещё и теги стандартов ID3v2, v3, v4, добавленные позже, отформатированные по-другому и находящиеся в начале файла (или, что более редко — в середине файла). Программы для тегирования часто добавляют как ID3v1, так и дополнительные (и более трудные для чтения) теги в начало файла. Для наших целей мы сосредоточимся только на файлах, содержащих корректные теги ID3v1 и ID3v1.1.

Теперь, когда мы знаем, как читать файл, мы можем перейти к различным способам записи файла.

Разные способы записи файлов

Запись в файл включает в себя достаточно много таких же операций, как и чтение файла. Рассмотрим их подробнее.

Запись списка термов в файл

Предположим, что мы хотим создать файл, который мы сможем прочитать функцией `file:consult`. Стандартная библиотека вообще-то не содержит такой функции, так что мы напишем свою собственную. Назовём эту функцию `unconsult`.

HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl unconsult(File, L) -> {ok, S} =  
file:open(File, write), lists:foreach(fun(X) -> io:format(S, "~p.~n", [X]) end, L), file:close(S).
```

Мы можем выполнить это из оболочки Эрланга, чтобы создать файл, называемый `test1.dat`:

```
1> lib_misc:unconsult("test1.dat", [{cats,["zorrow","daisy"]}, {weather,snowing}]). ok
```

Удостоверимся, что это действительно ОК: 2> `file:consult("test1.dat"). {ok, [{cats, ["zorrow","daisy"]}, {weather,snowing}]}`

Чтобы реализовать `unconsult` мы открываем файл на запись и затем используем `io:format(S, "~p.~n", [X])` для записи термов в файл. `io:format` — это рабочая лошадка для создания форматированного вывода. Для выполнения форматированного вывода мы вызываем функцию:

@spec io:format(io:device(), Format, Args) -> ok io:device — это некое устройство ввода-вывода (которое было открыто в режиме записи), Format — это строка, содержащая коды форматирования, а Args — это список элементов для вывода.

Для каждого элемента из Args в строке формата должна присутствовать команда форматирования. Команды форматирования начинаются с тильды (~).

Вот некоторые наиболее часто используемые команды форматирования:

~n

Перевод строки. ~n достаточно умен, так что работает платформонезависимо — на Unix — выведет в поток вывода ASCII (10), а на Windows — ASCII (13, 10)

~p

Структурная распечатка аргумента

~s

Аргумент является строкой

~w

Вывод данных со стандартным синтаксисом. Используется для вывода термов Эрланга

У форматной строки есть масса аргументов, которые никто не будет запоминать в здравом уме. Как говорил Эйнштейн — для констант есть справочники. А для полного списка параметров формата есть руководство по модулю io. Я помню только ~p, ~s и ~n. Если вы начнёте с них, у вас не возникнет лишних проблем.

Лирическое отступление

Я соврал. Вам наверняка понадобится больше, чем просто ~p, ~s, ~n. Вот пара примеров:

Формат

Результат

```
io:format("~10s~n", ["abc"])
```

labcl

```
io:format("~10s~n", ["abc"])
```

labcl

```
io:format("~10.3.+s~n",["abc"])
```

```
l+++++++abcl
```

```
io:format("~10.10.+s~n",["abc"])
```

```
labc+++++++l
```

```
io:format("~10.7.+s~n",["abc"])
```

```
l+++abc++++l
```

Запись строк в файл

Это похоже на предыдущий пример — мы просто используем другие команды форматирования:

```
1> {ok, S} = file:open("test2.dat", write). {ok,<0.62.0>} 2> io:format(S, "~s~n", ["Hello  
readers"]). ok 3> io:format(S, "~w~n", [123]). ok 4> io:format(S, "~s~n", ["that's it"]). ok 5>  
file:close(S).
```

Это создаёт файл называемый test2.dat со следующим содержимым:

```
Hello readers 123 that's it
```

Запись всего файла целиком одной операцией

Это наиболее эффективный способ записи в файл. Функция `file:write_file(File, IO)` записывает данные IO (который является списком ввода-вывода, т. е. списком, элементами которого могут быть другие списки ввода-вывода, бинарные данные, целые числа от 0 до 255) в файл File. При записи список автоматически плющится (делается плоским — flattened, т. е. все квадратные скобки устраняются). Этот способ крайне эффективен и я этим частенько пользуюсь. Программа в следующей части демонстрирует это.

Вывод URL-ов из файла

Давайте напишем простенькую функцию, называемую `urls2htmlFile(L, File)`, которая берет список URL-ов L и создаёт HTML файл, где URL-ы представлены в виде кликабельных ссылок. Это позволит нам отработать технику создания целого файла одной-единственной операцией ввода-вывода.

Мы поместим нашу программу в модуль `scavenge_url`.

HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"scavengeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"urlsHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"erl -module(scavenge_urls).
-export([urls2htmlFile/2, bin2urls/1]). -import(lists, [reverse/1, reverse/2, map/2]).
```

urls2htmlFile(Urls, File) -> file:write_file(File, urls2html(Urls)).

bin2urls(Bin) -> gather_urls(binary_to_list(Bin), []).

В программе две точки входа. urls2htmlFile(Urls, File) берёт список URL-ов и создаёт HTML файл, содержащий кликабельные ссылки для каждого URL. bin2urls(Bin) ищет по бинарным данным и возвращает список всех URL-ов, содержащихся в этих данных. Вот urls2htmlFile:

HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"scavengeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"urlsHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"erl urls2html(Urls) ->
[h1("Urls" ),make_list(Urls)].
```

h1(Title) -> ["

" , Title, "
\\n"].

make_list(L) -> ["

\\n" , map(fun(I) -> ["

- " ,I,"

\\n"] end, L), "

\\n"].

Этот код возвращает вложенный список символов. Заметьте, что мы не делали

попыток сплющить список (что было бы довольно неэффективно). Мы создали вложенный список символов и просто отправили его в функцию вывода. Когда мы записываем вложенный список в файл функцией `file:write_file` система ввода-вывода автоматически плющит список (т. е. записывает только символы из списка, но не скобки, создающие структуры списка). Ну и в конце — код, извлекающий URL-ы из бинарных данных:

HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"scavengeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"urlsHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/scavenge_urls.erl"erl gather_urls(" {Url, T1} =
collect_url_body(T, reverse(" gather_urls(T, L); gather_urls([], L) -> L.
```

```
collect_url_body(" ++ T, L) -> {reverse(L, "" ), T}; collect_url_body([HIT], L) ->
collect_url_body(T, [HIL]); collect_url_body([], _) -> {[], []}.
```

Чтобы выполнить это, нам надо иметь данные для разбора. Входные данные (бинарные данные) — это содержимое HTML страницы, так что нам нужна HTML страница для очистки от мусора. Для этого мы используем `socket_examples:nano_get_url` (см. главу 14.1, извлечение данных с сервера). Будем делать это по шагам в оболочке Эрланга:

```
1> B = socket_examples:nano_get_url("www.erlang.org"), L = scavenge_urls:bin2urls(B),
scavenge_urls:urls2htmlFile(L, "gathered.html"). ok
```

Это создаст файл `gathered.html`:


HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/gathered.html"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/gathered.html" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/gathered.html"gatheredHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/gathered.html".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/gathered.html"html
```

Urls

- [Older news.....](#)
- `<a href="http://www.erlang-consulting.com/training_fs.html"`

here

- [Megaco home](#)
- [Erlang Public License \(EPL\)](#)
-
- [download statistics graphs](#)
- [Erlang/OTP Test Server](#)
- [proceedings](#)
- [Read more in the release highlights.](#)
- 

Запись файлов с произвольным доступом

Запись в файл с произвольным доступом подобна чтению. Сначала мы должны открыть файл в режиме записи. Затем мы используем `file:pwrite(Position, Bin)` для записи в файл. Вот пример:

```
1> {ok, S} = file:open("...", [raw,write,binary]) {ok, ...} 2> file:pwrite(S, 10, <<"new">>) ok 3> file:close(S) ok
```

Этот код записывает символы "new", начиная со смещения 10 в файле, перезаписывая имеющееся содержимое файла.

Операции над директориями

Для операций над директориями в модуле `file` есть три функции. `list_dir(Dir)` используется для получения списка файлов в `Dir`, `make_dir(Dir)` создаёт новую директорию и `del_dir(Dir)` удаляет директорию.

Если мы выполним `list_dir` в директории с кодом, которую я использую при написания этой книги, то мы увидим следующее:

```
1> cd("/home/joe/book/erlang/Book/code"). /home/joe/book/erlang/Book/code ok 2> file:list_dir("."). {ok,["id3_v1.erl\~", "update_binary_file.beam", "benchmark_assoc.beam", "id3_v1.erl", "scavenge_urls.beam", "benchmark_mk_assoc.beam", "benchmark_mk_assoc.erl", "id3_v1.beam", "assoc_bench.beam", "lib_misc.beam", "benchmark_assoc.erl", "update_binary_file.erl", "foo.dets", "big.tmp", ..
```

Заметьте, что файлы в списке никак не упорядочены, никак не видно признаков, что данное имя является файлом или директорией, нет длин, вообще ничего нет.

Чтобы найти больше информации об индивидуальном файле в директории мы

используем функцию `file:read_file_info`, которая подробнее описывается в следующей части.

Поиск информации о файле

Для нахождения информации о файле `F` мы вызываем функцию `file:read_file_info(F)`. Она возвращает `{ok, Info}`, если `F` — это правильное имя файла или директории. `Info` — это запись (record) типа `#file_info`, которая определена так:

`-record(file_info,`

`{`

`Size,`

Размер файла в байтах

`Type,`

Атом: `device`, `directory`, `regular`, `other`

`Access,`

Атом: `read`, `write`, `read_write`, `none`

`Atime,`

Локальное время последнего чтения файла

`Mtime,`

Локальное время последней записи файла

`Ctime,`

Интерпретация этого поля зависит от операционной системы. В Unix это время последнего изменения файла или inode. В Windows — это время создания файла

`Mode,`

Целое число: права на файл. В Windows права владельца будут дублироваться для группы и пользователя

`Links,`

Количество ссылок на файл (1, если файловая система не поддерживает ссылки)

major_device,

Целое число: показывает файловую систему (в Unix) или номер устройства (A: = 0, B: = 1) Windows

Замечание: поля прав (mode) и доступа (access) перекрываются. Вы можете использовать права, чтобы установить несколько файловых атрибутов одной операцией. Впрочем, вы можете использовать access для простых операций.

Чтобы найти длину и тип файла мы вызываем функцию read_file_info (заметьте, что нам приходится подключать file.hrl, который содержит определение записи #file_info):

HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl -
include_lib("kernel/include/file.hrl"). file_size_and_type(File) -> case file:read_file_info(File) of
{ok, Facts} -> {Facts#file_info.type, Facts#file_info.size}; _ -> error end.
```

Теперь можно слегка улучшить вид списка, выведенного функцией list_file, добавив информацию о файлах в функции ls():

HYPERLINK

```
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"DownloadHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"libHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"miscHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/lib_misc.erl"erl ls(Dir) -> {ok, L} =
file:list_dir(Dir), map(fun(I) -> {I, file_size_and_type(I)} end, sort(L)).
```

Теперь список отсортирован и вдобавок содержит полезную информацию:

```
1> lib_misc:ls("."). [{"Makefile",{regular,1244}}, {"README",{regular,1583}}, {"abc.erl",
{regular,105}}, {"alloc_test.erl",{regular,303}}, ... {"socket_dist",{directory,4096}}, ...
```

Дополнительное удобство в том, что модуль filelib экспортирует несколько маленьких функций, таких как file_size(File) и is_dir(X). Это просто интерфейсы к file:read_file_info. Если нам надо всего лишь размер файла, то проще вызвать filelib:file_size, чем

`file:read_file_info` и распаковывать элементы записи `#file_info`.

Копирование и удаление файлов

`file:copy(Source, Destination)` копирует файл `Source` в файл `Destination`.

`file:delete(File)` удаляет файл `File`.

Всякая всячина

К текущему моменту мы упомянули ряд функций, которые я ежедневно использую для манипулирования файлами. И крайне редко мне приходится обращаться к документации за дополнительной информацией. Что же я пропустил такого, что может вам понадобиться? Я приведу краткий обзор основных вещей. А за подробными деталями обращайтесь к документации.

Режим файла: когда мы открываем файл функцией `file:open`, мы открываем его в определённом режиме или комбинацией режимов. Вообще-то есть много разных режимов. К примеру, можно открыть файл на чтение и запись сжатого `gzip` файла. Ну и т. д. Полный список как обычно находится в документации.

Время модификации, группы, ссылки: мы можем установить всё это функциями из `file`.

Коды ошибок: я опрометчиво сказал, что у всех ошибок вид `{error, Why}`. На самом деле `Why` — это атом (к примеру, `enoent` означает, что файл не существует и т. д.) - есть большое количество кодов ошибок и все они описаны в документации.

`filename`: модуль `filename` содержит некоторые полезные функции для сбора полных имён файлов и директорий, поиска расширений файлов и прочего, а также для построения имён файлов из компонентов пути. Всё это делается платформонезависимым образом.

`filelib`: модуль `filelib` содержит небольшое количество функций, которые помогают сэкономить нам время. Например, `filelib:ensure_dir(Name)` обеспечивает, что все родительские директории для данного файла или директории существуют, создавая их при необходимости.

Программа поиска

И как финальный пример, мы используем `file:list_dir` и `file:read_file_info` для создания программы поиска общего назначения.

Главная точка входа в этот модуль следующая:

`lib_find:files(Dir, RegExp, Recursive, Fun, Acc0)`

Аргументы для неё:

`Dir` — имя директории, откуда начинать поиск файла

`RegExp` — регулярное выражение для проверки имени найденного файла. Если файлы, которые мы встретим, совпадают с этим регулярным выражением, то вызывается функция `Fun(File, Acc)`, где `File` — это имя файла, которое успешно сопоставлено с регулярным выражением.

`Recursive = true | false` — это признак, который определяет будет ли поиск заходить в поддиректории текущей директории.

`Fun(File, AccIn) -> AccOut` — это функция, которая применяется к файлу, если имя файла соответствует регулярному выражению `RegExp`. Начальное значение аккумулятора `Acc` — это `Acc0`. Каждый раз, когда вызывается `Fun`, она должна вернуть новое значение аккумулятора, которое будет передано в `Fun` при следующем вызове этого `Fun`. Конечное значение аккумулятора — это значение, возвращаемое из функции `lib_find:files/5`.

Мы можем передать в `lib_find:files/5` любую функцию, какую только захотим. Например, мы можем построить список файлов, используя следующую функцию, передавая ей в начале пустой список:

```
fun(File, Acc) -> [File|Acc] end
```

Точка входа модуля `lib_find:files(Dir, ShelRegExp, Flag)` обеспечивает упрощённый вызов для более общего использования программы. `ShelRegExp` здесь — это упрощённое регулярное выражение, которое легче записать, чем полную форму регулярного выражения.

Пример такой короткой формы записи:

```
lib_find:files(Dir, "*.erl" , true)
```

рекурсивно ищет все файлы Эрланга, начиная с `Dir`. Если бы последний аргумент был `false`, то программа искала бы файлы Эрланга только в директории `Dir`, но не спускалась в поддиректории.

Итак, код:

[HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"](http://media.pragprog.com/titles/jaerlang/code/lib_find.erl)Download[HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"](http://media.pragprog.com/titles/jaerlang/code/lib_find.erl) [HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"](http://media.pragprog.com/titles/jaerlang/code/lib_find.erl)lib[HYPERLINK](http://media.pragprog.com/titles/jaerlang/code/lib_find.erl)

```
"http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"_HYPERLINK  
"http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"findHYPERLINK  
"http://media.pragprog.com/titles/jaerlang/code/lib_find.erl".HYPERLINK  
"http://media.pragprog.com/titles/jaerlang/code/lib_find.erl"erl -module(lib_find). -export([files/3,  
files/5]). -import(lists, [reverse/1]).
```

```
-include_lib("kernel/include/file.hrl" ).
```

```
files(Dir, Re, Flag) -> Re1 = regexp:sh_to_awk(Re), reverse(files(Dir, Re1, Flag, fun(File, Acc)  
->[File|Acc] end, [])).
```

```
files(Dir, Reg, Recursive, Fun, Acc) -> case file:list_dir(Dir) of {ok, Files} -> find_files(Files, Dir,  
Reg, Recursive, Fun, Acc); {error, _} -> Acc end.
```

```
find_files([File|T], Dir, Reg, Recursive, Fun, Acc0) -> FullName = filename:join([Dir,File]), case  
file_type(FullName) of regular -> case regexp:match(FullName, Reg) of {match, , } -> Acc =  
Fun(FullName, Acc0), find_files(T, Dir, Reg, Recursive, Fun, Acc); _ -> find_files(T, Dir, Reg,  
Recursive, Fun, Acc0) end; directory -> case Recursive of true -> Acc1 = files(FullName, Reg,  
Recursive, Fun, Acc0), find_files(T, Dir, Reg, Recursive, Fun, Acc1); false -> find_files(T, Dir,  
Reg, Recursive, Fun, Acc0) end; error -> find_files(T, Dir, Reg, Recursive, Fun, Acc0) end;  
find_files([], , , , A) -> A.
```

```
file_type(File) -> case file:read_file_info(File) of {ok, Facts} -> case Facts#file_info.type of  
regular -> regular; directory -> directory; _ -> error end; _ -> error end.
```