Глава 6. Компилирование и запуск ваших программ.

В предыдущих главах, мы мало говорили о компиляции и запуске программ - мы просто использовали оболочку Эрланг. Это вполне нормально для небольших примеров. Но по мере роста сложности ваших программ, вы, несомненно, захотите как-то автоматизировать этот процесс, чтобы упростить себе жизнь. Вот здесь и появляются make-файлы.

Существует три разных способа запуска ваших программ. И в этой главе, мы рассмотрим их все, так что вы сможете выбрать тот, который наиболее подходит к вашей ситуации.

Иногда могут происходить сбои: make-файлы могут не срабатывать, переменные окружения быть неправильными, также как и пути поиска файлов. Мы поможем вам разобраться с подобными проблемами, подсказав, что делать, если что-то пошло не так.

6.1 Запуск и остановка Эрланг оболочки (shell)

На UNIX системах (Включая Mac OS X) вы можете запустить оболочку Эрланга из командной строки консоли:

$ **erl**

Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1 (abort with \^G)

1>

В системе Microsoft Windows вам надо кликнуть на иконке Эрланга.

Простейшим способом остановить систему является нажатие Ctrl+C (в Виндоус - Ctrl+Break) и далее - A , как в следующем примере:

BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded

(v)ersion (k)ill (D)b-tables (d)istribution

a

$

Также, вместо этого, вы можете выполнить в оболочке Эрланга (или в программе) инструкцию erlang:halt() , что приведет к тому-же результату.

erlang:halt() - это BIF которая немедленно останавливает оболочку Эрланга и именно этим способом я и пользуюсь в большинстве случаев. Но, тем не менее, в этом способе есть определенное неудобство. Если вы запустили большое приложение работающее с базами данных и просто остановите всю систему, то при следующем запуске вам придется проходить через процесс восстановления после ошибки. Поэтому лучше останавливать систему более аккуратным способом.

Для контролируемой остановки, если оболочка реагирует на команды, вы можете набрать:

1> **q().**

ok

$

Тогда будут корректно закрыты все открытые файлы, остановлены базы данных (если они запущены) и закрыты все OTP приложения в установленном порядке. Команда q() это другое имя для команды init:stop() .

Если ни один из этих методов не работает, прочитайте раздел 6.6 *Как выбраться из неприятностей*.

6.2. Изменение окружения

Когда вы начинали программировать на Эрланге, вы, возможно, складывали все файлы и все модули в одну директорию из которой и стартовали сам Эрланг. В этом случае Эрланг без проблем мог найти ваш код. Но по мере роста сложности ваших приложений, вы наверняка захотите разделить его в управляемые куски и разместить их в различных директориях. А если вы будете включать в ваш проект внешний код, то он будет иметь свою собственную структуру директорий.

Установка пути поиска для загрузки кода.

Система исполнения приложений Эрланг использует механизм автозагрузки кода. Чтобы он работал корректно вы должны правильно установить несколько путей поиска, чтобы находилась именно правильная версия вашего кода.

Механизм загрузки кода на самом деле также написан на Эрланг - подробнее мы поговорим об этом в разделе E.4 *Динамическая загрузка кода* . И загрузка кода осуществляется , как это называют, "по требованию".

Когда система пытается вызвать функцию в модуле, который еще не был загружен, возникает исключение, и система пытается найти файл объектного кода пропущенного модуля. Если, например, этот пропущенный модуль называется myMissingModule, то

загрузчик кода, первым делом, будет пытаться найти файл с именем MyMissingModule.beam во всех директориях, которые входят в текущий путь загрузки кода. Поиск останавливается на первом таком найденном файле и объектный код из этого файла загружается в систему.

Вы можете узнать текущий путь загрузки кода запустив Эрланг и набрав команду code:get_path(). Вот пример ее работы:

code:get_path().

[".",

"/usr/local/lib/erlang/lib/kernel-2.11.3/ebin",

"/usr/local/lib/erlang/lib/stdlib-1.14.3/ebin",

"/usr/local/lib/erlang/lib/xmerl-1.1/ebin",

"/usr/local/lib/erlang/lib/webtool-0.8.3/ebin",

"/usr/local/lib/erlang/lib/typer-0.1.0/ebin",

"/usr/local/lib/erlang/lib/tv-2.1.3/ebin",

"/usr/local/lib/erlang/lib/tools-2.5.3/ebin",

"/usr/local/lib/erlang/lib/toolbar-1.3/ebin",

"/usr/local/lib/erlang/lib/syntax_tools-1.5.2/ebin",

...]

Следующие две функции наиболее часто используются для работы с путем загрузки кода:

@spec code:add_patha(Dir) => true | {error, bad_directory}

Добавляет новую директорию Dir в начало пути загрузки кода.

@spec code:add_pathz(Dir) => true | {error, bad_directory}

Добавляет новую директорию Dir в конец пути загрузки кода.

Часто это совершенно не важно какую из них использовать. Но надо следить за случаями, когда add_patha и add_pathz приводят к различным результатам. Если вы подозреваете, что загрузился не тот модуль, то вы можете набрать code:all_loaded() (которая выводит все загруженные модули) или code:clash() ("столкновение, коллизия")

чтобы разобраться, что же именно произошло не так.

Есть, также, еще несколько процедур в модуле code для работы с путем загрузки, но возможно они вам никогда не понадобятся, если только вы не станете писать весьма странные системные программы.

Обычной практикой является размещение всех этих команд в файле .erlang в вашей домашней директории. Либо же вы можете запускать Эрланг командой следующего вида:

```
erl -pa Dir1 -pa Dir2 ... -pz DirK1 -pz DirK2
```

где флаг -pa DirX добавляет директорию DirX в начало пути поиска кода, а -pz DirY добавляет директорию DirY в конец пути поиска кода.

Выполнение набора команд при старте системы Эрланг.

Мы уже познакомились, как можно установить путь поиска кода через команды в файле .erlang в вашей домашней директории. Но на самом деле вы можете поместить в этот файл любые команды Эрланг и, когда вы запустите Эрланг, он первым делом прочитает и выполнит все команды из этого файла.

Предположим мой файл .erlang выглядит следующим образом:

io:format("Running Erlang\~n").

code:add_patha(".").

code:add_pathz("/home/joe/2005/erl/lib/supported").

code:add_pathz("/home/joe/bin").

Тогда, при старте системы я увижу следующий вывод на консоль:

$ **erl**

Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Running Erlang

Eshell V5.5.1 (abort with \^G)

1>

Если в той директории, откуда стартует Эрланг также имеется файл .erlang то исполняться будет именно он, а не файл в вашей домашней директории. Таким образом вы можете настроить поведение Эрланга в зависимости от того, где он начал

свою работу. Это может быть полезно для специализированных приложений. В таком случае, вероятно, будет неплохой идеей добавить в такой файл несколько команд печати соответствующих сообщений, поскольку, в противном случае,вы можете забыть, что это локальный файл начальной настройки системы, что может привести к затруднениям в работе.

Подсказка: В некоторых системах, иногда бывает не совсем очевидно, а где именно находится ваша домашняя директория. Чтобы определить, что думает Эрланг, где она находится, сделайте следующее:

1> **init:get_argument(home).**

{ok,[["/home/joe"]]}

Откуда мы можем понять, что Эрланг думает, что нашей домашней директорией является /home/joe .

6.3 Различные способы запустить вашу программу

Программы Эрланга хранятся в модулях. как только вы написали вашу программу вы должны ее скомпилировать перед тем как запускать ее. Однако вы можете запустить вашу программу без компиляции запустив escript .

В следующих разделах мы покажем как скомпилировать и запустить несколько программ разными способами. Программы немного различаются, равно как и способы их запуска и остановки.

Первая программа hello.erl просто печатает фразу "Hello world!" . Она не будет запускать или останавливать систему Эрланг и ей не нужен доступ к аргументам в командной строке. Но наша вторая программа fac , наоборот будет нуждаться в доступе к параметру командной строки при ее запуске.

Вот наша простейшая программа, сохраненная в файле hello.erl. Она печатает строку "Hello World" с последующим переходом на новую строку (символ \~n интерпретируется как новая строка в модулях Эрланга io и io_lib)

```
-module(hello).

-export([start/0]).

start() ->

io:format("Hello world\~n" ).
```

Давайте скомпилируем и запустим эту программу тремя разными способами.

Компиляция и запуск в оболочке Эрланг (shell)

$ **erl**

Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1 (abort with \^G)

1> **c(hello).**

{ok,hello}

2> **hello:start().**

Hello world

ok

Компиляция и запуск из командной строки

$ **erlc hello.erl**

$ **erl -noshell -s hello start -s init stop**

Hello world

$

Быстрое исполнение

Часто возникает необходимость выполнить некоторую функцию Эрланга из командной строки операционной системы**.** В этом случае ключ -eval с аргументом бывает очень полезен для такого быстрого исполнения.

Приведем пример:

erl -eval 'io:format("Memory: \~p\~n" , [erlang:memory(total)]).'\

-noshell -s init stop

Пользователи Microsoft Windows: Чтобы это сработало, вам надо либо добавить в переменную PATH директорию содержащую исполняемый код системы Эрланг, либо вызывать erlc и erl полным именем (включая двойные кавычки). Например:

"C:\Program Files\erl5.5.3\bin\erlc.exe" hello.erl

..

Первая строка приведенная в рамочке (erlc hello.erl) компилирует файл hello.erl в файл

объектного кода hello.beam. Вторая команда возможна в трех различных вариантах:

-noshell

Запускает систему Эрланг без интерактивной оболочки (то есть вы не увидите стандартной надписи при запуске Эрланга).

-s hello start

Запускается функция hello:start().

Примечание: При использовании -s Module сам Module должен быть уже скомпилированным.

-s init stop

Когда закончится исполнение apply(hello, start, []) система автоматически выполнит функцию init:stop().

Команда erl -noshell ... может использоваться в командных файлах интерпретатора команд ОС (shell scripts), так что типичным случаем является создание командных файлов, которые устанавливают пути поиска (с помощью -pa *Directory*) и запускают программу.

В нашем примере мы использовали две -s команды. Число их в командной строке не ограничено. Каждая -s команда будет выполнена с помощью функции apply , а когда она закончится, начнется выполнение следующей команды.

Вот пример запуска hello.erl (файл hello.sh):

```
#!/bin/sh

erl -noshell -pa /home/joe/2006/book/JAERANG/Book/code\

-s hello start -s init stop
```

Примечание: Этот скрипт нуждается в абсолютном пути до директории, где находится файл hello.beam . Так что, хотя он и работает на моей машине, вам придется подредактировать его, чтобы он работал на вашей.

Чтобы запустить этот скрипт вы должны установить ему соответственно атрибуты командой chmod (только один раз) и тогда, будет можно его запускать:

```
$ chmod u+x hello.sh

$ ./hello.sh
```

Hello world

$

Примечание: В Microsoft Windows, прием #! не работает. Там придется создать .bat файл и использовать полный путь к исполняемой части системы Эрланг, если переменная PATH для нее не установлена соответственно.

Типичный скрипт-файл для Microsoft Windows может выглядеть приблизительно так: (файл hello.bat)

"C:\Program Files\erl5.5.3\bin\erl.exe" -noshell -s hello start -s init stop

Запуск через Escript

Используя escript вы можете запускать свои программы именно как скрипты, без их предварительной компиляции.

Предупреждение: escript входит в Эрланг начиная с версии R11B-4 и далее. Если у вас более ранняя версия Эрланга, то вы должны ее обновить до последней версии системы Эрланг.

Для запуска hello как скрипта, мы создадим следующий файл: (файл hello)

#!/usr/bin/env escript

main(_) ->

io:format("Hello world\n").

Экспорт функций во время разработки

Когда вы разрабатываете код, вам может быть немного неудобно все время возвращаться к разделу объявления экспорта функций, только для того, чтобы была возможность запустить их в оболочке Эрланг**.**

Специальная декларация для компилятора -compile(export_all) указывает ему экспортировать все функции в модуле. Это существенно упрощает жизнь пока вы разрабатываете код.

Когда вы закончили разработку кода в этом модуле, вы должны закомментировать декларацию export_all и добавить более точные декларации по экспорту функций. На это есть две причины. Во-первых, когда вы придете в следующий раз читать ваш код, вы будете знать что только важные функции были проэкспортированы наружу, а остальные функции не могут быть вызваны снаружи, так что вы можете их менять, как

вам это нужно, сохраняя лишь их интерфейс к проэкспортированным функциям. А во-вторых, компилятор может произвести гораздо более лучший код, когда он точно знает, какие именно функции проэкспортированы из данного модуля.

На UNIX системах мы можем запустить этот файл немедленно и без всякой компиляции:

chmod u+x hello

$ **./hello**

Hello world

$

Примечание: Параметры данного файла должны быть переведены в исполнимые (что в UNIX достигается командой chmod u+x *File*), что нужно сделать только один раз, а не каждый раз при запуске программы.

Программы с аргументами в командной строке

"Hello world" не имеет никаких аргументов. Давайте повторим наше упражнение для программы, которая вычисляет факториалы. Ей потребуется один аргумент.

Во-первых, вот ее код (файл fac.erl):

```
-module(fac).

-export([fac/1]).

fac(0) -> 1;

fac(N) -> N*fac(N-1).
```

Мы можем скомпилировать fac.erl и запустить его в оболочке Эрланга следующим образом:

$ **erl**

Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]

Eshell V5.5.1 (abort with \^G)

1> **c(fac).**

{ok,fac}

2> **fac:fac(25).**

15511210043330985984000000

Если же мы хотим запускать эту программу из командной строки, то нам надо ее модифицировать, чтобы она воспринимала оттуда параметр :

(файл fac1.erl)

```
-module(fac1).

-export([main/1]).

main([A]) ->

I = list_to_integer(atom_to_list(A)),

F = fac(I),

io:format("factorial \~w = \~w\~n" ,[I, F]),

init:stop().

fac(0) -> 1;

fac(N) -> N*fac(N-1).
```

Теперь мы можем ее скомпилировать и запустить:

$ **erlc fac1.erl**

$ **erl -noshell -s fac1 main 25**

factorial 25 = 15511210043330985984000000

Примечание: То что функция называется main - это не важно. Она может называться как угодно. важно только чтобы ее имя и имя функции в командной строке совпадали.

Наконец, мы можем запустить ее как скрипт:

(файл factorial)

```
#!/usr/bin/env escript

main([A]) ->

I = list_to_integer(A),

F = fac(I),

io:format("factorial \~w = \~w\~n" ,[I, F]).
```

```
fac(0) -> 1;

fac(N) ->

N * fac(N-1).
```

Компиляция здесь не нужна, просто запускаем его:

```
$ ./factorial 25

factorial 25 = 15511210043330985984000000

$
```

6.4 автоматическая компиляция в make-файлах.

Когда я пишу большую программу, я хочу автоматизировать этот процесс настолько, насколько это вообще возможно. На это есть две причины. Во-первых, в долгосрочной перспективе, это спасает от набивания опять и опять одних и тех-же команд во время многократного тестирования моей программы из множества строк и я не хочу чтобы мои пальцы у меня отвалились от всего этого.

Во-вторых, я часто откладываю текущий проект и вынужден поработать над чем-то еще. Могут пройти месяцы, прежде чем я вернусь к отложенному проекту и я совершенно забываю, как надо обрабатывать в нем код, что не оставляет мало шансов на его спасение!

make - это утилита, которая автоматизирует мою работу. Я использую ее для компиляции и распределения моего Эрланг - кода. Большинство моих make-файлов чрезвычайно просты и у меня есть для них готовые шаблоны, которые решают абсолютное большинство моих проблем.

Я не буду рассказывать о make-файлах в общем (см., например: http://en.wikipedia.org/wiki/Make). Вместо этого я покажу их форму, которая кажется мне полезной для компиляции Эрланг программ. В частности, мы рассмотрим маке-файлы, прилагаемые к данной книге, что значит, что вы будете понимать их и уметь писать, на их основе, собственные маке-файлы.

Шаблон маке-файла

Вот шаблон, на основе которого я создаю большинство своих маке-файлов (файл Makefile.template)

```
# leave these lines alone
```

```
.SUFFIXES: .erl .beam .yrl

.erl.beam:

erlc -W $<

.yrl.erl:

erlc -W $<

ERL = erl -boot start_clean

# Here's a list of the erlang modules you want compiling

# If the modules don't fit onto one line add a \ character

# to the end of the line and continue on the next line

# Edit the lines below

MODS = module1 module2 \

module3 ... special1 ...\

...

moduleN

# The first target in any makefile is the default target.

# If you just type "make" then "make all" is assumed (because

# "all" is the first target in this makefile)

all: compile

compile: ${MODS:%=%.beam} subdirs

## special compilation requirements are added here

special1.beam: special1.erl

${ERL} -Dflag1 -W0 special1.erl

## run an application from the makefile

application1: compile

${ERL} -pa Dir1 -s application1 start Arg1 Arg2
```

the subdirs target compiles any code in

sub-directories

subdirs:

cd dir1; make

cd dir2; make

...

remove all the code

clean:

rm -rf *.beam erl_crash.dump

cd dir1; make clean

cd dir2; make clean

Данный маке-файл начинается с правил компиляции модулей Эрланга и файлов с расширением .yrl (эти файлы содержат определения для программы Эрланг парсер-генератор)

Важной частью маке-файла является следующая строка:

MODS = module1 module2

Это список всех Эрланг - модулей, которые я хочу скомпилировать.

Каждый модуль из строки MODS будет скомпилирован командой erlc *Mod*.erl . Некоторые модули могут требовать специального обхождения с ними (например модуль special1 в файле-шаблоне), поэтому для них есть отдельное правило обработки.

Внутри маке-файла имеется много *целей*. Цель - это символьно-числовая строка, начинающаяся в первой позиции строки маке-файла и оканчивающаяся двоеточием (:) . В нашем маке-файле шаблоне all, compile и special.beam все являются целями. Чтобы выполнить маке-файл вы даете следующую команду в консоли:

$ **make [Target]**

Параметр Target - не обязательный. Если его нет, тогда подразумевается первая цель в файле. В нашем примере - это цель all, если другой цели не было задано в командной строке.

Если я хочу перестроить пою программу и запустить application1 , тогда я отдам команду make application1 . Если я хочу чтобы это было поведением по умолчанию, вызываемым только по команде make , то мне надо передвинуть строки определяющие цель application1 так, чтобы они стали первой целью в маке-файле.

Цель clean удаляет все скомпилированные объектные коды Эрланга и файл erl_crush.dump . Этот файл содержит информацию, которая может помочь отладить ваше приложение. Смотрите раздел 6.10 *Аварийный сброс системы (Crush Dump)* более детально по данной теме.

Специализация шаблона Make-файла

Я не сторонник беспорядка в моих программах, поэтому я обычно начинаю с шаблона маке-файла и удаляю из него все строки не имеющие отношения к моему приложению. Это дает мне маке-файл который короче и гораздо более понятный при его прочтении. Но, с другой стороны, вы можете иметь обобщенный, универсальный маке-файл, включаемый во все маке-файлы, поведение которого определяется их конкретными, специфическими переменными.

Результатом моего, вышеуказанного, процесса, может быть, например, нижеуказанный маке-файл:

```
.SUFFIXES: .erl .beam

.erl.beam:

erlc -W $<

ERL = erl -boot start_clean

MODS = module1 module2 module3

all: compile

${ERL} -pa '/home/joe/.../this/dir' -s module1 start

compile: ${MODS:%=%.beam}

clean:

rm -rf *.beam erl_crash.dump
```

6.5 Редактирование командами в оболочке Эрланга

Оболочка Эрланга содержит встроенный строковый редактор. Его команды являются подмножеством команд редактирования строчек используемых в популярном

редакторе emacs. Предыдущие строки могут быть вызваны и отредактированы несколькими командными символами. Они приведены далее (учтите что \^Key означает, что вы должны нажать Ctrl+Key):

Команда

**

\^A

\^E

\^F или стрелка вправо

\^B или стрелка влево

\^P или стрелка вверх

\^N или стрелка вниз

\^T

Табуляция

Описание

**

Начало строки

Конец строки

Вперед на символ

Назад на символ

Предыдущая строка

Следующая строка

Поменять последние два символа

Попытаться дописать имя текущего модуля или функции

6.6 Как выбраться из неприятностей

Эрланг иногда бывает трудно остановить. Перечислим здесь некоторые причины этого:

Оболочка Эрланга не отвечает

Обработка команды Ctrl+C была выключена

Эрланг был запущен с опцией -heart Cmd . Эта опция заставляет мониторинг процессов ОС следить за ОС-процессом Эрланга И если Эрланг-процесс в ОС умирает, то выполнить команду Cmd . Часто, при этом, Cmd просто перезапускает систему Эрланг. Это один из трюков, который используется для создания отказоустойчивых узлов Эрланга. Если вдруг слетит сам Эрланг (что, вообще говоря, не должно происходить), он просто будет перезапущен. Хитростью при остановке теперь Эрланга является найти сначала тот замеряющий пульс Эрланга процесс (используйте ps на UNIX-подобных машинах и Менеджер Задач (Task Manager) в Microsoft Windows и убить его перед тем как убивать процесс Эрланга в данной ОС.

Что-то может пойти совсем не так и оставить вас с непривязанным зомби-процессом Эрланга.

6.7 Когда что-то пошло не так

В этом разделе перечисляются некоторые общие проблемы и варианты их решения.

Неопределенный (Потерянный) код

Если вы пытаетесь запустить код в модуле, который загрузчик кода не может найти (поскольку заданный ему путь поиска некорректен) вы встретитесь с undef (неопределен) - сообщением об ошибке. Вот его пример:

1> **glurk:oops(1,23).**

** exited: {undef,[{glurk,oops,[1,23]},

{erl_eval,do_apply,5},

{shell,exprs,6},

{shell,eval_loop,3}]} **

На самом деле, здесь просто не существует модуля с именем glurk, но не это важно. Вы должны сконцентрироваться на рассмотрении сообщения об ошибке. Оно говорит нам, что система Эрланг пыталась вызвать функцию oops с аргументами 1 и 23 из модуля glurk . Следовательно, возможны четыре варианта, того что произошло при этом.

Возможно действительно не существует модуля glurk . Возможно его имя было указано слегка неверно (с опечаткой).

Кто нибудь видел мою точку с запятой?

Если вы забыли поставить точку с запятой между вариантами вызова вашей функции, или поставили туда точку, вместо этого, то у вас будут большие неприятности.

Если вы определили функцию foo/2 в строке 1234 своего модуля bar и поставили точку вместо точки с запятой, компилятор скажет вам:

bar.erl:1234 function foo/2 already defined.

Не делайте этого. Убедитесь, что ваши варианты функции всегда отделены друг от друга точкой с запятой.

Модуль glurk существует, но он не был скомпилирован. Система ищет файл с именем glurk.beam в директориях указанных ей в пути поиска кода.

Модуль glurk существует и он был откомпилирован, но директория, в которой находится glurk.beam не входит в путь поиска кода. Чтобы исправить эту ошибку вам, возможно, потребуется изменить путь поиска кода. Как это сделать мы рассмотрим чуть позже.

Существует несколько версий glurk в директориях поиска кода и мы выбрали не тот что нужно. Это редкая ошибка, но такое тоже возможно. Если вы подозреваете, что произошло именно это вы можете запустить функцию code:clash() , которая сообщает обо всех повторяющихся модулях в директориях входящих в путь поиска кода.

Мой маке-файл ничего не создает

Что может, вообще, случиться с маке-файлом? Ну, на самом деле, много чего. Но эта книга не про то как с ними работать, так что я ограничусь только самыми распространенными ошибками, связанными с ними.

Вот две самые частые ошибки с которыми я сталкивался:

Пробелы в маке-файле: Маке-файлы крайне привередливы. Хотя вы и не можете их видеть, но все связанные с предыдущей строки должны начинаться с символа табуляции (за исключением продолжения строк, где *на предыдущей строке* должен стоять в конце символ \ ). Если вдруг там окажутся пробелы, то маке-файл очень сильно будет смущен и вы отправитесь на поиски ошибки.

Пропущенный Эрланг-файл: Если отсутствует один из модулей указанный в цели маке-файла MODS , вы получите сообщение об ошибке. Для примера предположим, что MODS содержит модуль с именем glurk , но файла с именем glurk.erl нет в директории содержащей код. В этом случае маке выдаст следующую ошибку:

$ make

make: *** No rule to make target ‘glurk.beam',

needed by ‘compile'. Stop.

Возможно, также, что такой модуль присутствует, но, просто, его имя было указано некорректно в маке-файле.

Оболочка Эрланга не отвечает

Если оболочка не отвечает на команды, то могла произойти масса причин для этого. Сам процесс оболочки мог погибнуть, или вы могли запустить команду, которая никогда не закончится. Вы также могли забыть закрывающие кавычки или забыть набрать точка-возврат-каретки в конце вашей команды.

Но независимо от причин, вы можете прервать вашу оболочку Эрланга нажатием клавиш Ctrl+G и проследовать за следующим примером:

(1)

(2)

(3)

(4)

(5)

1> **receive foo -> true end.**

\^G

User switch command

--> **h**

c [nn] - connect to job

i [nn] - interrupt job

k [nn] - kill job

j - list all jobs

s - start local shell

r [node] - start remote shell

q - quit erlang

? l h - this message

--> **j**

1* {shell,start,[init]}

--> **s**

--> **j**

1 {shell,start,[init]}

2* {shell,start,[]}

--> c 2

Eshell V5.5.1 (abort with \^G)

1> **init:stop().**

ok

2> $

(1) Здесь я сказал оболочке принять сообщение foo . Но поскольку никто никогда не собирается посылать оболочке это сообщение, она входит в бесконечное ожидание этого сообщения. Я нажимаю Ctrl+G .

(2) Система входит в режим контроля работ (Job Control Mode или JCL). Тут я никогда не могу вспомнить его команды, поэтому я набираю h чтобы вызвать подсказку.

(3) Я набираю j чтобы вывести весь список работ. Работа номер 1 отмечена символом звездочка, что означает, что это оболочка Эрланга по умолчанию. Все прочие команды используют именно ее, если только им не указан параметр вида [nn] .

(4) Я набираю s чтобы запустить новую оболочку и опять набираю j . На этот раз я вижу что уже есть две оболочки с номерами 1 и 2 и оболочка с номером 2 стала оболочкой по умолчанию.

(5) Я набираю c 2 что подсоединяет меня к только что запущенной оболочке 2, в которой я останавливаю систему Эрланг.

Как видите у вас может быть множество оболочек в системе Эрланг, в которых можно набирать команды и переключаться между ними нажимая Ctrl+G . Вы даже можете запустить оболочку на удаленном Эрланг-узле с помощью команды r с соответствующим параметром.

6.8 Вызов помощи

В UNIX системах это делается так:

$ **erl -man erl**

NAME

erl - The Erlang Emulator

DESCRIPTION

The erl program starts the Erlang runtime system.

The exact details (e.g. whether erl is a script

or a program and which other programs it calls) are system-dependent.

...

Вы также можете получить справку по отдельным модулям:

$ **erl -man lists**

MODULE

lists - List Processing Functions

DESCRIPTION

This module contains functions for list processing.

The functions are organized in two groups:

...

Примечание: На UNIX системах страницы справки по умолчанию не установлены. Если команда erl - man не работает, то вам нужны страницы справки по Эрланг. Все они имеются в сжатом архиве по адресу http://www.erlang.org/download.html . Справочные страницы Эрланга должны быть разархивированы в корневой директории установки Эрланга (обычно это /usr/local/lib/erlang).

Документацию по Эрланг также можно загрузить в виде связанного множества HTML файлов. В ОС Microsoft Windows HTML документация устанавливается по умолчанию и доступна в разделе Эрланг в меню Старт данной ОС.

6.9 Настройка окружения

Оболочка Эрланга имеет множество встроенных команд. Вы можете прочитать о них всех с помощью команды оболочки help() :

1> **help().**

shell internal commands

b() -- display all variable bindings

e(N) -- repeat the expression in query

f() -- forget all variable bindings

f(X) -- forget the binding of variable X

h() -- history

...

Все эти команды определены в модуле shell_default .

Если вы хотите определить для оболочки свои собственные команды, просто создайте модуль с именем user_default . Например: (файл user_default.erl)

-module(user_default).

-compile(export_all).

hello() ->

"Hello Joe how are you?".

away(Time) ->

io:format("Joe is away and will be back in \~w minutes\~n" ,

[Time]).

Как только он будет скомпилирован и размещен где-то на пути вашего поиска кода, вы сможете вызывать любую его функцию без указания имени самого модуля:

1> **hello().**

"Hello Joe how are you?"

2> **away(10).**

Joe is away and will be back in 10 minutes

ok

6.10 Аварийный сброс данных системы (Crash Dump)

Если система Эрланг падает, то она оставляет после этого файл с именем erl-crash.dump . Содержимое этого файла может помочь вам разобраться, что же именно пошло не так. Чтобы его проанализировать есть анализатор аварий основанный на веб-интерфейсе. Чтобы его запустить наберите следующую команду:

1> **webtool:start().**

WebTool is available at http://localhost:8888/

Or http://127.0.0.1:8888/

{ok,<0.34.0>}

Потом укажите вашему браузеру адрес http://localhost:8888/ и вы сможете с удовольствием покопаться в журнале ошибки (error log) .

Итак, мы закончили рассмотрение низко-уровневых механизмов системы Эрланг и теперь можем перейти к параллельным программам. С этого момента вы вступаете на незнакомую территорию, но именно тут-то и начинается самое веселье.

Я не знаю можно ли запустить escript в Microsoft Windows. Если кто-то знает, как это сделать, напишите мне письмо об этом и я добавлю эту информацию в книгу.

Эрланговский парсер-генератор называется yecc (Эрланговская версия yacc - от "yet another compiler compiler" (еще один компилятор компиляторов)) Смотри руководство по нему в Интернете по адресу: http://www.erlang.org/contrib/parser_tutorial-1.0.tgz.