Глава 12 Интерфейсы с другими программами.

Предположим мы хотим связать программу на Эрланге с программой на C или Питоне или-же запустить из Эрланга командную консоль . Чтобы это сделать, нам надо запустить внешнюю программу в отдельном процессе операционной системы, отдельно от системы исполнения приложений (runtime) Эрланга и обмениваться с этой программой сообщениями через байт-ориентированный коммуникационный канал. Со стороны Эрланга, часть отвечающую за такую связь, называют *порт*. Процесс, который создает этот порт, называют процессом подключенным к этому порту. Подключенный процесс имеет особое значение для Эрланга: все сообщения к внешней программе помечаются PID-ом подключенного процесса, и все сообщения от внешней программы посылаются ему-же.

Взаимоотношения между подключенным процессом (C) портом (P) и внешним процессом операционной системы, показаны на Рисунке 12.1.

ЭСИП — Система исполнения приложений Эрланга

C — Процесс Эрланга подключенный к порту

P — порт.

Рисунок 12.1: Коммуникации с портом.

С точки зрения программиста порт ведет себя аналогично Эрланговскому процессу. Вы можете посылать ему сообщения можете зарегистрировать его (аналогично процессу) и так далее. Если внешняя программа обвалится (упадет), то сигнал exit будет послан подсоединенному к порту процессу, а если умрет подсоединенный процесс, то и внешняя программа будет убита (killed).

Возможно, вы удивитесь, почему все сделано именно так? Многие языки программирования позволяют просто подлинковывать программы на других языках программирования в свои исполняемые приложения. В Эрланге, мы этого не позволяем из соображений безопасности (исключение составляют подключаемые драйверы, которые мы обсудим далее в этой главе). Если бы мы просто подлинковали внешнюю программу в исполняемую среду Эрланга, то тогда ошибка в этой внешней программы могла бы легко повалить всю систему Эрланга. По этой причине, все другие языки должны исполняться снаружи системы Эрланга, в отдельном исполняемом процессе операционной системы. Система исполнения Эрланга и внешние процессы взаимодействуют (обмениваются информацией) через поток байтов.

12.1 Порты

Чтобы создать порт мы даем следующую команду:

Port = open_port(PortName, PortSettings)

Она возвращает нам порт. Следующие типы сообщений могут быть посланы порту (PidC означает PID подключенного к порту процесса):

Port!{PidC,{command,Data}} – посылает данные Data (байтовый список) порту.

Port!{PidC,{connect,Pid1}} – изменяет подключенный к порту процесс (с PID-ом PidC) на процесс с PID-ом Pid1.

Port!{PidC,close} – закрывает данный порт.

Подключенный к порту процесс может получать сообщения от внешней программы порта следующим стандартным образом:

receive

{Port, {data, Data} ->

...обработка данных, полученных от внешней программы...

В следующих разделах мы рассмотрим интерфейс Эрланга с очень простой С-программой, которая намеренно будет очень короткой, чтобы не отвлекать читателя от рассмотрения деталей интерфейса.

Замечание: Следующий пример намеренно выбран очень простым, чтобы можно было выделить в рассмотрении механизмы и протоколы общения с портом. Кодирование и декодирование сложных структур данных для обмена через протоколы порта — это сложная проблема, которую мы здесь не в состоянии решить. В конце этой главы мы дадим ссылки на некоторые библиотеки, которые могут быть использованы для построения интерфейсов к другим языкам программирования.

12.2 Интерфейс к внешней программе на С

Сначала мы посмотрим на нашу простейшую С-программу, выбранную для нашего примера:

/файл ports/example1.c /

```
int twice(int x){

return 2*x;

}
```

```
int sum(int x, int y){

return x+y;

}
```

Нашей конечной целью будет вызов этих C-функций из Эрланга. Мы хотим иметь возможность написать (на Эрланге):

```
X1 = example1:twice(23),

Y1 = example1:sum(45, 32),
```

То есть с точки зрения пользователя, example1 должен выглядеть, как обычный модуль Эрланга и, следовательно, все детали его взаимодействия (интерфейса) с программой на C, должны быть скрыты внутри модуля example1.erl

Нашему интерфейсу потребуется основная C-программа (main), которая будет декодировать данные присланные от Эрланг-программы. В нашем примере мы, в начале, определим протокол между портом и внешней C-программой. Он будет крайне простым и мы покажем как его реализовать на Эрланге и на C. Протокол будет следующим:

Все пакеты начинаются с двух-байтового кода их длины (Len) за которым будут следовать Len байтов данных.

Чтобы вызвать C-функцию twice(N) Эрланг-программа должна как-то закодировать такой вызов, согласованным образом (по договоренности). Мы будем полагать что это будет 2-х байтовая последовательность [1,N] , где 1 обозначает вызов вызов C-функции twice, а N — является ее одно-байтовым аргументом.

Для вызова C-функции sum(N,M) мы, аналогично, будем использовать байтовую последовательность [2,N,M] .

Предполагается, что возвращаемые значения будут длиной в 1 байт.

И программа на C, и программа на Эрланге должны следовать этому протоколу. Давайте, например, рассмотрим по шагам, что должно произойти, если Эрланг-программа захочет вычислить sum(45,32):

Порт посылает внешней программе байтовую последовательность 0,3,2,45,32 . Первые ее два байта представляют длину посылаемого пакета (3); следующий код 2 — означает вызов внешней функции sum; а 45 и 32 — это ее одно-байтовые аргументы.

Внешняя программа читает эти 5 байт из своего стандартного входного потока (input), вызывает функцию sum с переданными аргументами, и, потом, записывает байтовую

последовательность 0,1,77 в свой стандартный выходной поток (output). Первые два байта представляют из себя длину последующего пакета. А за ними следует однобайтовый результат работы функции sum(45,32) — 77.

Теперь нам надо написать программы на обоих сторонах интерфейса, которые строго следуют этому протоколу. Мы начнем с программы на C.

C-программа

Внешняя C-программа будет состоять из следующих трех файлов:

example1.c – он будет содержать функции, которые мы хотим вызывать.

example1_driver.c – здесь будет реализован байтовый протокол и вызываться нужные функции из example1.c .

erl_comm.c – здесь будут реализованы нужные процедуры чтения и записи буферов памяти.

example1_driver.c

```
/файл ports/example1_driver.c /

#include <stdio.h>

typedef unsigned char byte;

int read_cmd(byte *buff);

int write_cmd(byte *buff, int len);

int main() {

int fn, arg1, arg2, result;

byte buff[100];

while (read_cmd(buff) > 0) {

fn = buff[0];

if (fn == 1) {

arg1 = buff[1];

result = twice(arg1);

} else if (fn == 2) {
```

```
arg1 = buff[1];

arg2 = buff[2];

/* debug -- you can print to stderr to debug

fprintf(stderr,"calling sum %i %i\n",arg1,arg2); */

result = sum(arg1, arg2);

}

buff[0] = result;

write_cmd(buff, 1);

}

}
```

Этот код работает в бесконечном цикле, читая команды из стандартного входного потока (input), вызывает нужные функции и записывает результаты в стандартный выходной поток (output).

Если вы хотите использовать отладочную печать в C-программе, то вы должны направить ее вывод в stderr . Пример отладочной печати приведен в закомментированном участке кода программы.

erl_comm.c

```
/ erl_comm.c /

#include <unistd.h>

typedef unsigned char byte;

int read_cmd(byte *buf);

int write_cmd(byte *buf, int len);

int read_exact(byte *buf, int len);

int write_exact(byte *buf, int len);

int read_cmd(byte *buf)

{

int len;
```

```
if (read_exact(buf, 2) != 2)
return(-1);
len = (buf[0] << 8) | buf[1];
return read_exact(buf, len);
}
int write_cmd(byte *buf, int len)
{
byte li;
li = (len >> 8) & 0xff;
write_exact(&li, 1);
li = len & 0xff;
write_exact(&li, 1);
return write_exact(buf, len);
}
int read_exact(byte *buf, int len)
{
int i, got=0;
do {
if ((i = read(0, buf+got, len-got)) <= 0)
return(i);
got += i;
} while (got<len);
return(len);
}
int write_exact(byte *buf, int len)
```

```
{

int i, wrote = 0;

do {

if ((i = write(1, buf+wrote, len-wrote)) <= 0)

return (i);

wrote += i;

} while (wrote<len);

return (len);

}
```

Этот код предназначен для обработки пакетов с заголовком из 2-х байт, который будет соответствовать опции {packet,2} для программы порта драйвера (см. ниже).

Программа на Эрланге

Драйвер порта со стороны Эрланга обеспечивается следующим модулем:

/файл ports/example1.erl /

```
-module(example1).

-export([start/0, stop/0]).

-export([twice/1, sum/2]).

start() ->

spawn(fun() ->

register(example1, self()),

process_flag(trap_exit, true),

Port = open_port({spawn, "./example1" }, [{packet, 2}]),

loop(Port)

end).

stop() ->
```

```
example1 ! Stop.

twice(X) -> call_port({twice, X}).

sum(X,Y) -> call_port({sum, X, Y}).

call_port(Msg) ->

example1 ! {call, self(), Msg},

receive

{example1, Result} ->

Result

end.

loop(Port) ->

receive

{call, Caller, Msg} ->

Port ! {self(), {command, encode(Msg)}},

receive

{Port, {data, Data}} ->

Caller ! {example1, decode(Data)}

end,

loop(Port);

stop ->

Port ! {self(), close},

receive

{Port, closed} ->

exit(normal)

end;

{'EXIT', Port, Reason} ->
```

```
exit({port_terminated,Reason})

end.

encode({twice, X}) -> [1, X];

encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.
```

Порт открывается следующей командой:

```
Port = open_port({spawn, "./example1"}, [{packet, 2}])
```

Опция {packet,2} говорит системе автоматически добавлять к пакетам, адресованным удаленной программе, 2-х байтовый заголовок длины этого пакета. Поэтому, если мы пошлем сообщение {PidC,{command,[2,45,32]}} порту, то драйвер этого порта добавит 2-х байтовую длину в заголовок пакета и пошлет последовательность 0,3,2,45,32 внешней программе.

При приеме данных, порт также будет полагать, что каждый входящий пакет предваряется 2-х байтовым заголовком и будет удалять его байты до того как передать данные подключенному к порту процессу Эрланга.

Давайте соберем наши программы. Мы используем для этого следующий make-файл для их построения. Команда make example1 собирает внешнюю программу, которая (ее имя) используется как аргумент в Эрланговской функции open_port . Заметьте, что данный make-файл также включает в себя код для построения прилинкованного драйвера, который будет рассмотрен далее в этой главе.

Make-файл

/файл ports/Makefile /

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

.erl.beam:

erlc -W $<

MODS = example1 example1_lid

all: ${MODS:%=%.beam} example1 example1_drv.so

example1: example1.c erl_comm.c example1_driver.c

gcc -o example1 example1.c erl_comm.c example1_driver.c
```

example1_drv.so: example1_lid.c example1.c

gcc -o example1_drv.so -fpic -shared example1.c example1_lid.c

clean:

rm example1 example1_drv.so *.beam

Запуск программы

Теперь мы можем запустить нашу программу:

1> **example1:start().**

<0.32.0>

2> **example1:sum(45, 32).**

77

4> **example1:twice(10).**

20

...

На чем мы и завершим наш первый пример.

Но, до того как мы перейдем к следующему разделу, мы должны отметить следующее:

В данной программе не делается попыток унификации представления C и Эрланга о том, что такое есть целое число. Мы просто полагаем, что целые в Эрланге и C у нас это просто однобайтовые числа и игнорируем все возможные проблемы точности представления и знаков. В реальных приложениях, нам придется гораздо серьезнее задуматься над типами и их представлениями для передаваемых данных. Это может быть не простым вопросом, поскольку Эрланг свободно манипулирует целыми числами произвольной размерности, в то время как такие языки как C имеют различные фиксированные представления для целых определенных размерностей и так далее.

Мы не можем просто запустить Эрланг-функции, без предварительного запуска драйвера, который отвечает за интерфейс (то есть, какая-то программа должна до этого выполнить example1:start() , прежде чем мы сможем запустить нашу программу). Нам бы, конечно, хотелось, чтобы это происходило автоматически, во время старта нашей системы. Это вполне возможно, но для этого требуются некоторые знания на тему того, как система стартует и останавливается. Мы рассмотрим эти вопросы позже в разделе 18.7 Приложения (Эрланга).

2.3 open_port

В предыдущем разделе мы использовали функцию open_port без подробного рассказа каковы бывают ее аргументы и что они, при этом, делают. Мы видели только использование open_port с аргументом {packet,2}, который добавляет и убирает 2-х байтовый заголовок — длину пакета для данных, пересылаемых между Эрлангом и внешней программой. На самом деле у open_port может быть довольно много других аргументов.

Некоторые, из наиболее используемых, из них приведены далее:

@spec open_port(PortName, [Opt]) -> Port

PortName может иметь следующие виды:

{spawn, Command}

Запускает внешнюю программу. Command — имя этой внешней программы, которая запускается вне рабочего пространства Эрланга, если только не найдется прилинкованный драйвер с именем Command.

{fd, In, Out}

Позволяет Эрланговскому процессу получить доступ к любому открытому файловому дескриптору, который использует Эрланг. Файловые дескриптор In может быть использован для стандартного ввода, а файловый дескриптор Out — для стандартного вывода (см пример подключения стандартного ввода и вывода по ссылке: HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"httpHYPERLINK "http://www.erlang.org/examples/examples-2.0.html"://HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"wwwHYPERLINK "http://www.erlang.org/examples/examples-2.0.html".HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"erlangHYPERLINK "http://www.erlang.org/examples/examples-2.0.html".HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"orgHYPERLINK "http://www.erlang.org/examples/examples-2.0.html"/HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"examplesHYPERLINK "http://www.erlang.org/examples/examples-2.0.html"/HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"examplesHYPERLINK "http://www.erlang.org/examples/examples-2.0.html"-2.0.HYPERLINK "http://www.erlang.org/examples/examples-2.0.html"html).

Опции Opt могут быть следующими:

{packet, N}

Пакеты будут предваряться N-байтовым (N=1,2,4) счетчиком байт в пакете данных.

stream

Сообщения пересылаются без подсчета их длины. Приложение должно само уметь обрабатывать такие пакеты данных.

{line, Max}

Обмен сообщениями на основе принципа по одному-в-строке. Если строка более чем Max байт, то она разбивается после Max байт на следующую строку (и так далее).

{cd, Dir}

Действует только для параметра {spawn, Command}. Внешняя программа запускается в директории Dir .

{env, Env}

Действует только для параметра {spawn, Command}. Переменные окружения для внешней программы расширяются переменными из списка Env, состоящего из пар вида {VarName, Value} ({ИмяПеременной, ЕеЗначение}), где VarName и Value — это строки.

Это не полный список аргументов для функции open_port . Их полное описание можно найти в документации для модуля erlang .

12.4 Подключаемые драйверы

Иногда возникает потребность, чтобы программа написанная на другом языке работала внутри системы исполнения приложений Эрланга. В этом случае, программа пишется как разделяемая библиотека, которая динамически подлинковывается к системе исполнения Эрланга. Подключаемые драйверы выглядят для программиста так же, как и программы порта и подчиняются точно тем-же протоколам, что и они.

Создание подключаемых драйверов — это самый эффективный путь взаимодействия с кодом на другом языке из Эрланга, но он, также, и самый опасный. Любая фатальная ошибка в подключенном драйвере повалит всю систему исполнения приложений Эрланга со всеми запущенными в ней процессами. По этой причине, использование подключаемых драйверов не рекомендуется; и их следует использовать только тогда, когда все возможные альтернативы не подходят.

Чтобы проиллюстрировать этот подход, мы превратим использованную нами ранее программу в подключаемый драйвер. Чтобы это сделать нам потребуется три файла:

example1_lid.erl – это Эрланг сервер.

example1.c – содержит C-функции, которые мы хотим использовать. Ничем не отличается от использованного нами ранее.

example1_lid.c – это C-программа, которая вызывает C-функции из example1.c

Код Эрланга, поддерживающий такой интерфейс приведен далее:

/файл ports/example1_lid.erl /

```
-module(example1_lid).

-export([start/0, stop/0]).

-export([twice/1, sum/2]).

start() ->

start("example1_drv" ).

start(SharedLib) ->

case erl_ddll:load_driver("." , SharedLib) of

ok -> ok;

{error, already_loaded} -> ok;

_ -> exit({error, could_not_load_driver})

end,

spawn(fun() -> init(SharedLib) end).

init(SharedLib) ->

register(example1_lid, self()),

Port = open_port({spawn, SharedLib}, []),

loop(Port).

stop() ->

example1_lid ! stop.

twice(X) -> call_port({twice, X}).

sum(X,Y) -> call_port({sum, X, Y}).
```

```
call_port(Msg) ->

example1_lid ! {call, self(), Msg},

receive

{example1_lid, Result} ->

Result

end.

loop(Port) ->

receive

{call, Caller, Msg} ->

Port ! {self(), {command, encode(Msg)}},

receive

{Port, {data, Data}} ->

Caller ! {example1_lid, decode(Data)}

end,

loop(Port);

stop ->

Port ! {self(), close},

receive

{Port, closed} ->

exit(normal)

end;

{'EXIT', Port, Reason} ->

io:format("\~p \~n" , [Reason]),

exit(port_terminated)

end.
```

encode({twice, X}) -> [1, X];

encode({sum, X, Y}) -> [2, X, Y].

decode([Int]) -> Int.

Если мы сравним эту программу с ее предыдущей версией для интерфейса порта, мы увидим, что они практически идентичны.

Программа подключаемого драйвера состоит большей частью из кода работающего с элементами его структуры driver. Команда make example1_drv.so для make-файла, приведенного нами ранее, позволяет построить нужную разделяемую библиотеку данного драйвера.

/файл ports/example1_lid.c /

/ *example1_lid.c* /

#include <stdio.h>

#include "erl_driver.h"

typedef struct {

ErlDrvPort port;

} example_data;

static ErlDrvData example_drv_start(ErlDrvPort port, char *buff)

{

example_data *d = (example_data*)driver_alloc(sizeof(example_data));

d->port = port;

return (ErlDrvData)d;

}

static void example_drv_stop(ErlDrvData handle)

{

driver_free((char*)handle);

}

static void example_drv_output(ErlDrvData handle, char *buff, int bufflen)

```
{
example_data d = (example_data)handle;
char fn = buff[0], arg = buff[1], res;
if (fn == 1) {
res = twice(arg);
} else if (fn == 2) {
res = sum(buff[1], buff[2]);
}
driver_output(d->port, &res, 1);
}
ErlDrvEntry example_driver_entry = {
NULL, / F_PTR init, N/A /
example_drv_start, / L_PTR start, called when port is opened /
example_drv_stop, / F_PTR stop, called when port is closed /
example_drv_output, /* F_PTR output, called when erlang has sent
data to the port */
NULL, /* F_PTR ready_input,
called when input descriptor ready to read*/
NULL, /* F_PTR ready_output,
called when output descriptor ready to write */
"example1_drv" , / char driver_name, the argument to open_port */
NULL, / F_PTR finish, called when unloaded /
NULL, / F_PTR control, port_command callback /
NULL, / F_PTR timeout, reserved /
NULL / F_PTR outputv, reserved /
```

```
};

DRIVER_INIT(example_drv) / must match name in driver_entry /

{

return &example_driver_entry;

}
```

Вот результаты работы этих программ:

```
1> c(example1_lid).

{ok,example1_lid}

2> example1_lid:start().

<0.41.0>

3> example1_lid:twice(50).

100

4> example1_lid:sum(10, 20).

30
```

12.5 Примечания

В этой главе мы рассмотрели как использовать порты для взаимодействия из Эрланга с внешними программами. В дополнение к протоколу порта, можно использовать еще несколько BIF для работы с ними. Все это описано в документации к модулю erlang .

Но сейчас у вас, возможно, возникает вопрос, а как передавать сложные структуры данных между Эрлангом и внешними программами? Как пересылать строки, кортежи и так далее? К сожалению, не существует простого ответа на этот вопрос и порт предоставляет только низкоуровневый протокол обмена последовательностями байтов между Эрлангом и внешним миром. Между прочим, точно такая же проблема существует и для socket-обмена данными. Сокеты обеспечивают только потоки байтов между двумя приложениями, а как их интерпретировать — полностью перекладывается на сами эти приложения.

Тем не менее, существует несколько библиотек, включенных в дистрибутивы Эрланга, которые облегчают проблемы общения Эрланговских программ с внешними программами. Они включают в себя следующие примеры:

http://www.erlang.org/doc/pdf/erl_interface.pdf

Интерфейс Erl (ei) – это набор C-функций и макросов для кодирования и декодирования Эрланговских форматов. На стороне Эрланга используется функция term_to_binary для представления Эрланговского терма (объекта) в виде байтовой последовательности. А на стороне C-программы, указанные функции из ei могут быть использованы для распаковки этих бинарных данных. Кроме того, ei можно использовать и для создания бинарных данных, которые на стороне Эрланга распаковываются с помощью binary_to_term .

http://www.erlang.org/doc/pdf/ic.pdf

IDL компилятор Эрланга (ic). Приложение ic – это реализация на Эрланге OMG IDL компилятора.

http://www.erlang.org/doc/pdf/jinterface.pdf

Jinterface – это набор средств обеспечивающих взаимодействие Джавы и Эрланга. Он обеспечивает полное отображение типов Эрланга в объекты Джавы, кодирование и декодирование Эрланговских термов, связь с процессами Эрланга и так далее, включая большой набор дополнительных средство и возможностей.