Глава 18

Создание системы с использованием OTP

В этой главе мы будем создавать систему, выполняющую функции сервера в интернет-компании. Наша компания имеет два объекта продажи: простые числа и области ?услуга вычисления площади?. Покупатели могут купить простое число у нас или мы вычислим область геометрического объекта для них. Я думаю, что наша компания имеет огромный потенциал.

Мы создадим два сервера: один будет генерировать простые числа, а второй вычислять площадь. Чтобы сделать это, мы будем использовать gen_server, о котором мы говорили в разделе 16.2, "Начнем с gen_server" на странице 301.

Когда мы создаем систему, мы должны думать об ошибках, которые могут возникнуть. Хотя мы тщательно тестируем свое приложение, некоторые ошибки могут ускользнуть из поля зрения. Так что будем предполагать, что один из наших серверов может иметь фатальную ошибку, которая обрушит наш сервер. На самом деле, мы специально совершим ошибку, в одном из серверов, которая будет приводить к аварии.

Для обнаружения факта обрушения сервера нам необходимо иметь соответствующий механизм, чтобы определить, что случилась авария и перезапустить сервер. Для этого мы используем идею *дерева супервизоров*. Мы создадим супервизора, который будет следить за нашими серверами и перезапускать их при авариях.

Конечно же, если сервер потерпел аварию, мы захотим знать причины аварии, чтобы в дальнейшем устранить обнаруженные проблемы. Для протоколирования всех ошибок мы будем использовать регистратор ошибок OTP (OTP error logger). Мы покажем, как настраивать регистратор ошибок и как генерировать отчет об ошибках по журналу ошибок.

При вычислении простых чисел, в частности больших простых чисел, наш процессор может перегреться. Для предотвращения перегрева нам потребуется включать мощный вентилятор. Чтобы сделать это, нам нужно подумать о системе оповещения - тревогах ?алармах?. Мы будем использовать подсистему обработки событий OTP для генерирования и обработки тревог ?алармов?.

Все эти задачи (создание сервера, надзор за сервером, регистрация ошибок и определение тревог) являются типичными проблемами, которые должны быть решены в любой системе промышленного масштаба. В общем, даже если наша компания имеет довольно мутную перспективу, мы сможем использовать эту архитектуру в других системах. На самом деле, такая архитектура используется в ряде успешных коммерческих компаний.

В итоге, когда все заработает, мы упакуем весь наш код в единое OTP приложение. Вкратце, это специализированная группировка всех частей задачи, которая позволяет системе OTP запускать, управлять и останавливать задачу.

Порядок, в котором изложен материал несколько замысловат и имеет обратные зависимости между различными частями. Регистрация ошибок представляет собой особый случай управления событиями. Тревоги - это просто события, а сам регистратор ошибок - это контролируемый процесс, хотя процесс супервизора и может вызывать функции регистратора ошибок.

Я попытаюсь все это упорядочить и представить части в некотором осмысленном порядке. Итак, мы будем делать следующее:

Рассмотрим идеи использования типичного обработчика событий.

Увидим как работает регистратор ошибок.

Добавим управление тревогами.

Напишем два сервера.

Создадим дерево надзора и добавим в него наши серверы.

Упакуем всё в единое приложение.

18.1 Типичная обработка событий

Событие - это когда что-нибудь происходит, что-нибудь заслуживающее внимания программиста, который думает о том, кто и что должен делать в данном случае.

Когда мы программируем и происходит что-нибудь заметное, мы просто отправляем сообщение о событии зарегистрированному процессу. Что-то вроде этого:

RegProcName ! {event, E}

E - это событие (любой Эрланг-элемент (term)). RegProcName - имя зарегистрированного процесса.

Нам не нужно заботиться о том, что происходит с сообщением, когда мы его отправили. Мы просто выполнили работу и сообщили о том, что что-то случилось.

Теперь переключим наше внимание на процесс приёма сообщений о событиях. Этот процесс называется "обработчик событий". Простейший возможный обработчик событий - это "ничего не делающий" обработчик. Когда он принимает сообщение {event, X}, он ничего не делает с этим событием; просто отбрасывает его в сторону.

Вот наша первая попытка создания программы типичного обработчика событий:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"СкачатьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"eventHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"handlerHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"erl

```
-module(event_handler).

-export([make/1, add_handler/2, event/2]).

%% make a new event handler called Name

%% the handler function is noOp -- so we do nothing with the event

make(Name) ->

register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).

add_handler(Name, Fun) -> Name ! {add, Fun}.

%% generate an event

event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->

receive

{add, Fun1} ->

my_handler(Fun1);

{event, Any} ->

(catch Fun(Any)),

my_handler(Fun)

end.

no_op(_) -> void.
```

API обработчика событий следующий:

event_handler:make(Name)

Приготовить "ничего не делающий" обработчик называемый Name (атом). Это то место, куда будут направляться события.

event_handler:event(Name, X)

Отправить событие X обработчику Name.

event_handler:add_handler(Name, Fun)

Добавить обработчик Fun к обработчику событий Name. Когда происходит событие X, обработчик выполнит Fun(X).

Теперь создадим обработчик и сгенерируем ошибку:

1> event_handler:make(errors).

true

2> event_handler:event(errors, hi).

{event,hi}

Ничего особенного не произойдет, потому что мы не подключили модуль обратных вызовов к этому обработчику.

Чтобы получить обработчик событий, который делает что-нибудь осмысленное, необходимо написать для него модуль обратных вызовов и подключить этот модуль к обработчику:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"motorHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"controllerHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"erl

```
-module(motor_controller).

-export([add_event_handler/0]).

add_event_handler() ->

event_handler:add_handler(errors, fun controller/1).
```

```
controller(too_hot) ->

io:format("Turn off the motor\~n" );

controller(X) ->

io:format("\~w ignored event: \~p\~n" ,[?MODULE, X]).
```

Скомпилируем этот код и подключим к обработчику:

```
3> c(motor_controller).

{ok,motor_controller}

4> motor_controller:add_event_handler().

{add,#Fun<motor_controller.0.99476749>}
```

Теперь, когда события будут отправлены обработчику, они будут обработаны функцией motor_controller:controller/1:

```
5> event_handler:event(errors, cool).

motor_controller ignored event: cool

{event,cool}

6> event_handler:event(errors, too_hot).

Turn off the motor

{event,too_hot}
```

И в чём же смысл проделанной работы? Во первых, мы задали имя, на которое будут отправляться события. В данном случае, это зарегистрированный процесс errors. Затем, мы определили протокол отправки событий зарегистрированному процессу. Но мы ничего не сказали о том, что происходит с событиями, которые получает этот процесс. На самом деле, всё что случается будет обработано в функции noOp(X). В конце мы подключим другой обработчик событий, но об этом позже.

Очень позднее связывание с "изменением ваших мыслей"

Предположим, что мы пишем функцию, которая скрывает конструкцию event_handler:event от программиста. Например, мы пишем следующее:

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

"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

too_hot() ->

event_handler:event(errors, too_hot).

В этом случае мы говорим программисту вызывать lib_misc:too_hot() в своем коде, когда дела пойдут плохо. В большинстве языков программирования вызов функции too_hot был бы статически или динамически прилинкован в код программы. Так как вызов прилинкован, значит он выполняет фиксированную работу зависящую от кода. Если позднее изменится наше понимание и мы решим изменить что-нибудь, то это будет не простой путь изменения нашей системы.

Подход Эрланга к обработке событий абсолютно другой. Он позволяет отделить генерирацию событий от обработки событий. Мы можем изменить обработку в любое время, просто передав новую функцию обработки в обработчик событий. Ничего не линкуется статически, и каждый обработчик может быть изменен тогда, когда вам это потребуется.

Используя такой механизм, мы можем построить систему *меняющуюся со временем* и не требующую остановки для замены кода.

Примечание: Это не "позднее связывание" - это "ОЧЕНЬ позднее связывание, дающее возможность думать так или иначе".

Возможно, вы немного запутались. Почему мы говорим об обработчиках событий? Ключевой момент повествования в том, что обработчик событий предоставляет нам инфраструктуру, в которую мы можем внедрять свои обработчики.

Инфраструктура регистратора ошибок строится из шаблона обработчика событий. Мы можем устанавливать различные обработчики в регистраторе ошибок для достижения различных целей.

18.2 Регистратор ошибок

Система OTP строится на настраиваемых регистраторах ошибок. Регистратор ошибок можно рассматривать с трех точек зрения. С точки зрения *программиста* - это вызовы функций позволяющие вести журнал ошибок. Точка зрения *конфигурации* - это то, как регистратор ошибок сохраняет данные. Точка зрения *отчетов* - это анализ ошибок

после того как они случились. Мы рассмотрим каждую из точек зрения.

Журналирование/Протоколирование ошибок

Что касается программиста, API регистратора ошибок достаточно прост. Вот он:

@spec error_logger:error_msg(String) -> ok

Отправить сообщение об ошибке регистратору ошибок.

1> error_logger:error_msg("An error has occurred\n").

=ERROR REPORT==== 28-Mar-2007::10:46:28 ===

An error has occurred

ok

@spec error_logger:error_msg(Format, Data) -> ok

Отправить сообщение об ошибке регистратору ошибок. Аргументы такие же как и для io:format(Format, Data).

2> error_logger:error_msg("~s, an error has occurred\n", ["Joe"]).

=ERROR REPORT==== 28-Mar-2007::10:47:09 ===

Joe, an error has occurred

ok

@spec error_logger:error_report(Report) -> ok

Отправить стандартный отчет об ошибке регистратору ошибок.

• @type Report = [{Tag, Data} | term()] | string() | term()]

• @type Tag = term()

• @type Data = term()

3> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).

=ERROR REPORT==== 28-Mar-2007::10:51:51 ===

tag1: data1

a_term

tag2: data

Это только небольшая часть доступного API. Обсуждение деталей не очень интересно. В наших программах мы будем использовать только error_msg. Полное описание можно посмотреть на страницах руководства по error_logger.

Настройка регистратора ошибок

Существует много способов настроить регистратор ошибок. Мы можем видеть все ошибки в окне оболочки Эрланга (это режим по-умолчанию, специально настраивать не требуется). Мы можем записывать все ошибки попадающие в окно оболочки в один отформатированный файл. И, наконец, мы можем создать кольцевой ?циклический? журнал ошибок. Можете думать о колцевом журнале как о большом кольцевом буфере, который содержит сообщения, выдаваемые регистратором ошибок. Новые сообщения записываются в конец журнала, а когда журнал полон, то записи из начала журнала удаляются.

Кольцевые журналы используются очень часто. Вам решать как много файлов журналов использовать и насколько они будут большими, а система сама позаботится об удалении старых и создании новых файлов в кольцевом буфере. Вы можете задать подходящий размер файла, чтобы сохранить в нем записи за несколько дней, этого обычно достаточно в большинстве случаев.

Стандарные регистраторы ошибок

Когда мы запускаем Эрланг, мы можем использовать аргумент boot:

$ erl -boot start_clean

Такой запуск обеспечит окружение для разработки программ. Будет поддерживаться только простая регистрация ошибок. (Команда erl без аргумена boot эквивалентна команде erl -boot start_clean)

$ erl -boot start_sasl

Такой запуск обеспечит окружение для запуска системы готовой к эксплуатации. Библиотеки поддержки системной архитектуры (SASL - System Architecture Support Libraries) позаботится о регистрации ошибок, о перегрузках системы и так далее.

Настройку журналов лучше всего делать из файлов настроек, потому что вряд ли кто-нибудь помнит все аргументы регистратора. Далее мы рассмотрим как работает система по-умолчанию и увидим четыре конфигурации, которые меняют поведение регистратора.

SASL без настройки

Вот что происходит, когда мы запускаем SASL без файла настроек:

```
$ erl -boot start_sasl

Erlang (BEAM) emulator version 5.5.3 [async-threads:0] ...

=PROGRESS REPORT==== 27-Mar-2007::11:49:12 ===

supervisor: {local,sasl_safe_sup}

started: [{pid,<0.32.0>},

{name,alarm_handler},

{mfa,{alarm_handler,start_link,[]}},

{restart_type,permanent},

{shutdown,2000},

{child_type,worker}]

... many lines removed ...

Eshell V5.5.3 (abort with \^G)
```

Сейчас мы вызовем одну из конструкций error_logger для отчета об ошибке:

```
1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error

ok
```

Заметим, что отчет об ошибке отобразился в оболочке Эрланга. Вывод отчетов об ошибках зависит от настроек регистратора ошибок.

Управление регистратором

Регистратор ошибок предоставляет несколько типов отчетов:

Отчеты супервизора

Отчеты о том, что OTP супервизор запускает или останавливает подчинённые процессы (мы поговорим о супервизорах а разделе 18.5 "Дерево надзора", на странице 351).

Отчеты о выполнении

Отчеты о запуске или остановке супрвизора.

Отчеты об авариях

Отчеты об остановке выполнения с сообщением о причине отказа помимо normal или shutdown.

Эти три типа отчетов обрабатываются автоматически и не требуют вмешательства программиста.

Мы можем дополнительно вызвать конкретную конструкцию модуля error_handler, чтобы обработать все три типа отчетов. Это позволит нам использовать соощения об ошибках, предупреждения и сообщения информационного характера. Три этих термина ничего не означают; воспринимайте их как теги, позволяющие программисту различать природу элементов в журнале ошибок.

Позже, когда журнал ошибок будет проанализирован, эти теги помогут нам решить какие из элементов журнала исследовать. Когда мы настраиваем регистратор ошибок, мы можем указать, что требуется сохранять только ошибки, а все остальные элементы игнорировать. Теперь давайте напишем файл настроек elog1.config для настройки регистратора ошибок:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"config

```
%% no tty

[{sasl, [

{sasl_error_logger, false}

]}].
```

Если мы запустим систему с этим файлом настроек, то будем получать только сообщения об ошибках, мы не получим сообщений о ходе выполнения и прочих. Все эти сообщения будут выводиться только в окно оболочки Эрланга.

```
$ erl -boot start_sasl -config elog1

1> error_logger:error_msg("This is an error\n").
```

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error

ok

Текстовый файл и оболочка (shell)

Следующий файл настроек выдаёт список ошибок в окно оболочки Эрланга и дублирует все сообщения из оболочки в файл:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"2.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"config

%% single text file - minimal tty

[{sasl, [

%% All reports go to this file

{sasl_error_logger, {file, "/home/joe/error_logs/THELOG" }}

]}].

Для проверки мы запустим Эрланг, сгенерируем сообщение об ошибке и посмотрим результат в файле:

$ erl -boot start_sasl -config elog2

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error ok

Если мы посмотрим файл /home/joe/error_logs/THELOG, в начале файла мы найдем следующие строки:

=PROGRESS REPORT==== 28-Mar-2007::11:30:55 ===

supervisor: {local,sasl_safe_sup}

started: [{pid,<0.34.0>},

{name,alarm_handler},

{mfa,{alarm_handler,start_link,[]}},

{restart_type,permanent},

{shutdown,2000},

{child_type,worker}]

...

Кольцевой журнал и оболочка

Эта конфигурация даст нам возможность выводить все ошибки в оболочку Эрланга плюс дублирование всего вывода оболочки в кольцевой журнальный файл. Такой вариант наиболее часто используется на практике.

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"elogHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"3.HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"config

%% rotating log and minimal tty

[{sasl, [

{sasl_error_logger, false},

%% задать параметры кольцевого журнала

%% директория с файлом журнала

{error_logger_mf_dir,"/home/joe/error_logs" },

%% # кол-во байт выделенное для журнала

{error_logger_mf_maxbytes,10485760}, % 10 MB

%% максимальное кол-во файлов-журналов

{error_logger_mf_maxfiles, 10}

]}].

$erl -boot start_sasl -config elog3

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 28-Mar-2007::11:36:19 ===

This is an error

false

При запуске системы все ошибки будут направляться в файл кольцевого журнала. Позже в этой главе мы рассмотрим как извлекать эти ошибки из файла-журнала.

Продуктивная среда ?!ПРОДАКШЕН!?

В продуктивной среде нам, на самом деле, интересны только отчеты об ошибках, а не о процессе выполнения или какая-либо информация, поэтому заставим регистратор отчитываться только об ошибках. Без этих настроек систему будут перегружать информационные отчеты и отчеты о процессе исполнения.

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"4.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"config

%% rotating log and errors

[{sasl, [

%% minimise shell error logging

{sasl_error_logger, false},

%% only report errors

{errlog_type, error},

%% define the parameters of the rotating log

%% the log file directory

{error_logger_mf_dir,"/home/joe/error_logs" },

%% # bytes per logfile

{error_logger_mf_maxbytes,10485760}, % 10 MB

%% maximum number of

{error_logger_mf_maxfiles, 10}

]}].

В результате запуска получим нечто похожее на предыдущий пример. С той лишь разницей, что регистрироваться будут только ошибки.

Анализируем ошибки

Чтение журнала ошибок входит в обязанности модуля rb. Этот модуль имеет чрезвычайно простой интерфейс.

1> rb:help().

Report Browser Tool - usage

---

rb:start() - start the rb_server with default options

rb:start(Options) - where Options is a list of:

{start_log, FileName}

- default: standard_io

{max, MaxNoOfReports}

- MaxNoOfReports should be an integer or 'all'
- default: all

...

... many lines omitted ...

...

Запустим браузер отчетов и скажем ему сколько записей из журнала читать (в данном случае последние двадцать):

2> rb:start([{max,20}]).

rb: reading report...done.

3> rb:list().

No Type Process Date Time

== ==== ======= ==== ====

11 progress <0.29.0> 2007-03-28 11:34:31

10 progress <0.29.0> 2007-03-28 11:34:31

9 progress <0.29.0> 2007-03-28 11:34:31

8 progress <0.29.0> 2007-03-28 11:34:31

7 progress <0.22.0> 2007-03-28 11:34:31

6 progress <0.29.0> 2007-03-28 11:35:53

5 progress <0.29.0> 2007-03-28 11:35:53

4 progress <0.29.0> 2007-03-28 11:35:53

3 progress <0.29.0> 2007-03-28 11:35:53

2 progress <0.22.0> 2007-03-28 11:35:53

1 error <0.23.0> 2007-03-28 11:36:19

ok

> rb:show(1).

ERROR REPORT <0.40.0> 2007-03-28 11:36:19

This is an error

ok

Для того чтобы найти конкретную ошибку мы можем использовать такую команду как rb:grep(RegExp), при её использовании найдется то, что описано в регулярном выражении RegExp. Я не хочу углубляться в то, как анализировать журналы ошибок. Лучше потратьте время и поинтересуйтесь модулем rb и все увидите сами. Замечу, что на самом деле вам никогда не потребуется удалять журналы ошибок, точный механизм кольцевых журналов в конце концов сам удалит старые записи.

Если вам требуется оставить все сообщения об ошибках, вы можете задать интервалы и удалять информацию по мере необходимости.

18.3 Управление тревогами

Когда мы пишем наше приложение, нам требуется только одна тревога - будем реагировать только тогда, когда начнет перегреваться процессор, потому что мы вычисляем гигантские простые числа (помните, мы создали компанию по продаже простых чисел). Вот теперь-то мы и будем использовать настоящий OTP-шный обработчик тревог (и не простой как в начале главы).

Обработчик тревог это модуль обратных вызовов OTP для поведения gen_event. Вот его код:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"ЗагрузатьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"myHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"*HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"alarmHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"*HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"handlerHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"erl

```
-module(my_alarm_handler).

-behaviour(gen_event).

%% gen_event callbacks

-export([init/1, handle_event/2, handle_call/2,

handle_info/2, terminate/2]).

%% init(Args) must return {ok, State}

init(Args) ->

io:format("*** my_alarm_handler init:\~p\~n" ,[Args]),

{ok, 0}.

handle_event({set_alarm, tooHot}, N) ->

error_logger:error_msg("*** Tell the Engineer to turn on the fan\~n" ),

{ok, N+1};

handle_event({clear_alarm, tooHot}, N) ->
```

```
error_logger:error_msg("*** Danger over. Turn off the fan\~n" ),

{ok, N};

handle_event(Event, N) ->

io:format("*** unmatched event:\~p\~n" ,[Event]),

{ok, N}.

handle_call(_Request, N) -> Reply = N, {ok, N, N}.

handle_info(_Info, N) -> {ok, N}.

terminate(Reason, N) -> ok.
```

Этот код очень похож на код обратных вызовов gen_server, который мы видели раньше в разделе 16.3, "*Что же происходит когда мы вызываем сервер*?", на странице 306. Интересующей нас конструкцией является handle_event(Event, State). Она возвращает {ok, NewState}. Event - это кортеж имеющий форму {EventType, EventArg}, где EventType это атом set_event или clear_event, а EventArg - это пользовательские аргументы. Чуть позже мы рассмотрим как генерируются такие события.

А теперь позабавимся. Мы запустим систему, сгенерируем тревогу, установим обработчик тревог, сгенерируем новую тревогу, и так далее:

```
$ erl -boot start_sasl -config elog3

1> alarm_handler:set_alarm(tooHot).

ok

=INFO REPORT==== 28-Mar-2007::14:20:06 ===

alarm_handler: {set,tooHot}

2> gen_event:swap_handler(alarm_handler,

{alarm_handler, swap},

{my_alarm_handler, xyz}).

*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}

3> alarm_handler:set_alarm(tooHot).

ok
```

=ERROR REPORT==== 28-Mar-2007::14:22:19 ===

*** Tell the Engineer to turn on the fan

4> alarm_handler:clear_alarm(tooHot).

ok

=ERROR REPORT==== 28-Mar-2007::14:22:39 ===

*** Danger over. Turn off the fan

Что же здесть присходит?

Мы запустили Эрланг с -boot start_sasl. Когда мы сделали это, мы получили стандартный обработчик тревог. Когда мы устанавливаем или очищаем тревогу, то ничего не происходит. Это простой "ничего не делающий" обработчик событий, мы такие рассматривали раньше.

Когда мы установили тревогу (строка 1), мы просто получили информационный отчет. Здесь нет специальной обработки тревог.

Мы установили свой обработчик тревог (строка 2). Аргумент в my_alarm_handler (xyz) не имеет особого значения; синтаксис требует какое-нибудь значение, но поскольку нам не требуются значения, мы просто используем атом xyz, мы сможем увидеть этот аргумент при выводе на консоль.

Строка •• my_alarm_handler_init: ... напечатана из нашего модуля обратных вызовов.

Мы установили и очистили тревогу tooHot (строки 3 и 4). Это отработал наш обработчик тревог. Мы можем проверить, прочитав вывод на консоли.

Чтение журнала

Давайте вернёмся обратно к регистратору ошибок и посмотрим, что там происходит:

1> rb:start([{max,20}]).

rb: reading report...done.

2> rb:list().

No Type Process Date Time

== ==== ======= ==== ====

...

3 info_report <0.29.0> 2007-03-28 14:20:06

2 error <0.29.0> 2007-03-28 14:22:19

1 error <0.29.0> 2007-03-28 14:22:39

3> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 14:22:39

*** Danger over. Turn off the fan

ok

4> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 14:22:19

*** Tell the Engineer to turn on the fan

Итак, здесь мы видим как работает механизм регистратора ошибок.

На практике мы должны были бы убедиться, что журнал ошибок достаточно велик для хранения данных за несколько дней или даже недель. Каждые несколько дней (или недель) мы бы проверяли журнал на предмет ошибок.

Примечание: Модуль rb содержит функции для выбора ошибок указанного типа и извлечения этих ошибок в файл. В результате процесс анализа ошибок может быть полностью автоматизирован.

18.4 Серверные приложения

Наше приложение состоит из двух серверов: сервер простых чисел и сервер рассчёта площади. Рассмотрим сервер простых чисел. Он написан с использованием поведения gen_server (см. раздел 16.2 "*Начинаем с gen_server*" на стр. 301). Замечу, что он включает в себя обработку тревог которую мы разработали в предыдущем разделе.

Сервер простых чисел

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"primeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"_HYPERLINK

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

```
-module(prime_server).

-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server callbacks

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,

terminate/2, code_change/3]).

start_link() ->

gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->

%% 20000 is a timeout (ms)

gen_server:call(?MODULE, {prime, N}, 20000).

init([]) ->

%% Note we must set trap_exit = true if we

%% want terminate/2 to be called when the application

%% is stopped

process_flag(trap_exit, true),

io:format("\~p starting\~n" ,[?MODULE]),

{ok, 0}.

handle_call({prime, K}, _From, N) ->

{reply, make_new_prime(K), N+1}.

handle_cast(_Msg, N) -> {noreply, N}.

handle_info(_Info, N) -> {noreply, N}.

terminate(Reason, N) ->
```

```
io:format("\~p stopping\~n" ,[?MODULE]),

ok.

code_change(OldVsn, N, Extra) -> {ok, N}.

make_new_prime(K) ->

if

K > 100 ->

alarm_handler:set_alarm(tooHot),

N = lib_primes:make_prime(K),

alarm_handler:clear_alarm(tooHot),

N;

true ->

lib_primes:make_prime(K)

end.
```

Сервер площади

Теперь рассмотрим сервер площади. Он так же построен на поведении gen_server. Заметьте, написание сервера очень быстрый процесс. Когда я писал этот пример, я просто скопировал код из сервера простых чисел и вставил его в новый сервер. Все заняло несколько минут.

Сервер площади не является идеальной программой и содержит преднамеренную ошибку (сможете ее найти?). Мой не очень коварный план - это заставить сервер рухнуть, чтобы быть перестрахованным супервизором. А потом получить отчёт обо всех ошибках в журнале ошибок.

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"areaHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"serverHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"erl

```
-module(area_server).

-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server callbacks

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,

terminate/2, code_change/3]).

start_link() ->

gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

area(Thing) ->

gen_server:call(?MODULE, {area, Thing}).

init([]) ->

%% Note we must set trap_exit = true if we

%% want terminate/2 to be called when the application

%% is stopped

process_flag(trap_exit, true),

io:format("\~p starting\~n" ,[?MODULE]),

{ok, 0}.

handle_call({area, Thing}, _From, N) -> {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N) -> {noreply, N}.

handle_info(_Info, N) -> {noreply, N}.

terminate(Reason, N) ->

io:format("\~p stopping\~n" ,[?MODULE]),

ok.

code_change(OldVsn, N, Extra) -> {ok, N}.

compute_area({square, X}) -> X*X;
```

compute_area({rectonge, X, Y}) -> X*Y.

надзор one_for_one

Если один процесс рухнет, он будет перезапущен

надзор all_for_one

Если один процесс рухнет, все процессы будут прерваны и перезапущены

18.1 Два вида дерева надзора

18.5 Дерево надзора

Дерево надзора - это дерево процессов. Самые верхние процессы (супервизоры) в дереве наблюдают за нижними (рабочими) процессами в дереве и перезапускют нижние процессы, если те аварийно завершаются. Два вида дерева надзора вы можете увидеть на рисунке 18.1.

One-for-one дерево надзора

В надзоре one-for-one, если один процесс рухнул, то супервизор рестартует

только этот процесс.

All-for-one дерево надзора

В надзоре all-for-one, если любой из процессов рухнет, то все поднадзорные

процессы будут уничтожены (вызовом функции terminate/2 в соответстующем

модуле обратных вызовов). Затем все рабочие процессы будут рестартованы.

Супервизоры создаются с использованием OTP поведения *supervisor*. Это поведение описывается в специальном модуле обратных вызовов, который содержит стратегию надзора и правила запуска отдельных рабочих процессов в дереве надзора. Дерево надзора определяется функцией следующего вида:

init(...) ->

{ok, {RestartStrategy, MaxRestarts, Time},

[Worker1, Worker2, ...]}.

Здесь RestartStrategy это один из атомов one_for_one или all_for_one. MaxRestarts и Time указывают на "частоту перезапуска". Если супервизор перезапускает процессы большее число раз, чем указано в MaxRestarts за Time секунд, то работа супервизора

будет прервана. Это делается для того, чтобы остановить бесконечный цикл перезапуска процессов, если они содержат ошибки и останавливаются из-за них.

Worker1, Worker2 и т.д. это кортеж описывающий как запускать каждый из рабочих процессов. Мы увидим, как это выглядит уже скоро.

Теперь давайте вернемся к нашей компании и создадим дерево надзора.

Для начала, думаю, нам надо выбрать имя для нашей компании. Пусть будет sellaprime. Задача супервизора sellaprime - это конечно же держать всегда запущенными сервер простых чисел и сервер площади. Для этого напишем уже другой модуль обратных вызовов, теперь для gen_supervisor. Вот этот модуль:

HYPERLINK ""ЗагрузитьHYPERLINK "" HYPERLINK ""sellaprimeHYPERLINK ""_HYPERLINK ""supervisorHYPERLINK "".HYPERLINK ""erl

```
-module(sellaprime_supervisor).

-behaviour(supervisor). % see erl -man supervisor

-export([start/0, start_in_shell_for_testing/0, start_link/1, init/1]).

start() ->

spawn(fun() ->

supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])

end).

start_in_shell_for_testing() ->

{ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = []),

unlink(Pid).

start_link(Args) ->

supervisor:start_link({local,?MODULE}, ?MODULE, Args).

init([]) ->

%% Install my personal error handler

gen_event:swap_handler(alarm_handler,

{alarm_handler, swap},
```

{my_alarm_handler, xyz}),

{ok, {{one_for_one, 3, 10},

[{tag1,

{area_server, start_link, []},

permanent,

10000,

worker,

[area_server]},

{tag2,

{prime_server, start_link, []},

permanent,

10000,

worker,

[prime_server]}

]}}.

Самая важная часть - это структура данных возвращаемая функцией init/1:

HYPERLINK ""ЗагрузитьHYPERLINK "" HYPERLINK ""sellaprimeHYPERLINK ""_HYPERLINK ""supervisorHYPERLINK "".HYPERLINK ""erl

{ok, {{one_for_one, 3, 10},

[{tag1,

{area_server, start_link, []},

permanent,

10000,

worker,

[area_server]},

{tag2,

{prime_server, start_link, []},

permanent,

10000,

worker,

[prime_server]}

]}}.

Эта структура данных определяет стратегию надзора. Мы говорили о стратегии надзора и частоте перезапуска выше. Сейчас осталось дать определение для сервера областей и сервера простых чисел.

Определение для Worker процессов имеет следующий вид:

{Tag, {Mod, Func, ArgList},

Restart,

Shutdown,

Type,

[Mod1]}

Что же обозначают все эти аргументы?

Tag

Атом, который будет использоваться для ссылки на рабочий процесс в дальнейшем (если потребуется).

{Mod, Func, ArgList}

Определение функции, которую супервизор будет использовать для запуска рабочего процесса. Оно используется как аргумент при вызове apply(Mod, Fun, ArgList).

Restart = permanent | transient | temporary

permanent - процесс будет перезапускаться всегда. transient - процесс будет перезапущен только, если получено ненормальное значение при выходе. temporary - процесс запускается только один раз и не перезапускается.

Shutdown

Время остановки. Это максимально разрешенное время для остановки рабочего процесса. Если время остановки процесса будет превышено, то процесс просто будет убит. (Возможны и другие значения - см. руководство по Супервизору)

Type = worker | supervisor

Тип надзираемого процесса. Мы можем сконструировать дерево надзора над супервизорами, добавляя процесс супервизора вместо рабочего процесса.

[Mod1]

Это имя модуля обратных вызовов, если дочерний процесс имеет поведение supervisor или gen_server (Возможны и другие значения - см. руководство по Супервизору)

18.6 Запуск системы

Теперь мы готовы первый раз запустить нашу компанию. Мы вернулись. Кто хочет купить первое простое число?

Давайте запустим систему:

$ erl -boot start_sasl -config elog3

1> sellaprime_supervisor:start_in_shell_for_testing().

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

Теперь сделаем правильный запрос:

2> area_server:area({square,10}).

100

Сейчас сделаем неправильный запрос:

3> area_server:area({rectangle,10,20}).

area_server stopping

=ERROR REPORT==== 28-Mar-2007::15:15:54 ===

** Generic server area_server terminating

** Last message in was {area,{rectangle,10,20}}

Действительно ли работает стратегия надзора?

Эрланг был разработан для программирования отказоустойчивых систем. Первоначальная разработка была сделана в Лаборатории Вычислительной Техники Шведской компании Эрикссон. С тех пор группа OTP вела разработку с помощью десятков сотрудников компании. Используя gen_server, gen_supervisor и другие поведения Эрланга строились системы с надежностью 99.9999999% (тут девять девяток). При правильном использовании, механизм обработки ошибок может помочь сделать вашу программу работающей вечно (ну, или почти вечно). Регистратор ошибок, описанный здесь, работает уже в течение нескольких лет в реальных, живых продуктах.

** When Server state == 1

** Reason for termination ==

** {function_clause,[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]}

area_server starting

** exited: {{function_clause,

[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]},

{gen_server,call,

[area_server,{area,{rectangle,10,20}}]}} **

Упс - что же тут случилось? Сервер площади рухнул; мы умышленно допустили в коде ошибку. Авария была обнаружена супервизором и сервер был перезапущен. Все это запротоколировал регистратор ошибок.

После аварии, все вернулось к нормальному состоянию, как и должно было. Давайте сейчас сделаем правильный запрос:

4> area_server:area({square,25}).

625

У нас все опять работает. Теперь давайте сгенерируем маленькое простое число:

5> prime_server:new_prime(20).

Generating a 20 digit prime .....

37864328602551726491

А теперь сгенерируем большое простое число:

6> prime_server:new_prime(120).

Generating a 120 digit prime

=ERROR REPORT==== 28-Mar-2007::15:22:17 ===

*** Tell the Engineer to turn on the fan

.....

=ERROR REPORT==== 28-Mar-2007::15:22:20 ===

*** Danger over. Turn off the fan

765525474077993399589034417231006593110007130279318737419683

288059079481951097205184294443332300308877493399942800723107

Теперь у нас работоспособная система. Если на сервере случится авария, то он автоматически будет перезапущен, а регистратор ошибок проинформирует нас об этом.

Сейчас давайте рассмотрим журнал ошибок:

1> rb:start([{max,20}]).

rb: reading report...done.

rb: reading report...done.

{ok,<0.53.0>}

2> rb:list().

No Type Process Date Time

== ==== ======= ==== ====

20 progress <0.29.0> 2007-03-28 15:05:15

19 progress <0.22.0> 2007-03-28 15:05:15

18 progress <0.23.0> 2007-03-28 15:05:21

17 supervisor_report <0.23.0> 2007-03-28 15:05:21

16 error <0.23.0> 2007-03-28 15:07:07

15 error <0.23.0> 2007-03-28 15:07:23

14 error <0.23.0> 2007-03-28 15:07:41

13 progress <0.29.0> 2007-03-28 15:15:07

12 progress <0.29.0> 2007-03-28 15:15:07

11 progress <0.29.0> 2007-03-28 15:15:07

10 progress <0.29.0> 2007-03-28 15:15:07

9 progress <0.22.0> 2007-03-28 15:15:07

8 progress <0.23.0> 2007-03-28 15:15:13

7 progress <0.23.0> 2007-03-28 15:15:13

6 error <0.23.0> 2007-03-28 15:15:54

5 crash_report area_server 2007-03-28 15:15:54

4 supervisor_report <0.23.0> 2007-03-28 15:15:54

3 progress <0.23.0> 2007-03-28 15:15:54

2 error <0.29.0> 2007-03-28 15:22:17

1 error <0.29.0> 2007-03-28 15:22:20

Что-то тут не так. У нас есть отчет об аварии сервера областей. Как узнать, что случилось (если бы мы не знали об этом)?

9> rb:show(5).

CRASH REPORT <0.43.0> 2007-03-28 15:15:54

Crashing process

pid <0.43.0>

registe

Глава 18

Создание системы с использованием OTP

В этой главе мы будем создавать систему, выполняющую функции сервера в интернет-компании. Наша компания имеет два объекта продажи: простые числа и области ? услуга вычисления площади?. Покупатели могут купить простое число у нас или мы вычислим область геометрического объекта для них. Я думаю, что наша компания имеет огромный потенциал.

Мы создадим два сервера: один будет генерировать простые числа, а второй вычислять площадь. Чтобы сделать это, мы будем использовать gen_server, о котором мы говорили в разделе 16.2, "Начнем с gen_server" на странице 301.

Когда мы создаем систему, мы должны думать об ошибках, которые могут возникнуть. Хотя мы тщательно тестируем свое приложение, некоторые ошибки могут ускользнуть из поля зрения. Так что будем предполагать, что один из наших серверов может иметь фатальную ошибку, которая обрушит наш сервер. На самом деле, мы специально совершим ошибку, в одном из серверов, которая будет приводить к аварии.

Для обнаружения факта обрушения сервера нам необходимо иметь соответствующий механизм, чтобы определить, что случилась авария и перезапустить сервер. Для этого мы используем идею *дерева супервизоров*. Мы создадим супервизора, который будет следить за нашими серверами и перезапускать их при авариях.

Конечно же, если сервер потерпел аварию, мы захотим знать причины аварии, чтобы в дальнейшем устранить обнаруженные проблемы. Для протоколирования всех ошибок мы будем использовать регистратор ошибок OTP (OTP error logger). Мы покажем, как настраивать регистратор ошибок и как генерировать отчет об ошибках по журналу ошибок.

При вычислении простых чисел, в частности больших простых чисел, наш процессор может перегреться. Для предотвращения перегрева нам потребуется включать мощный вентилятор. Чтобы сделать это, нам нужно подумать о системе оповещения -

тревогах ?алармах?. Мы будем использовать подсистему обработки событий OTP для генерирования и обработки тревог ?алармов?.

Все эти задачи (создание сервера, надзор за сервером, регистрация ошибок и определение тревог) являются типичными проблемами, которые должны быть решены в любой системе промышленного масштаба. В общем, даже если наша компания имеет довольно мутную перспективу, мы сможем использовать эту архитектуру в других системах. На самом деле, такая архитектура используется в ряде успешных коммерческих компаний.

В итоге, когда все заработает, мы упакуем весь наш код в единое OTP приложение. Вкратце, это специализированная группировка всех частей задачи, которая позволяет системе OTP запускать, управлять и останавливать задачу.

Порядок, в котором изложен материал несколько замысловат и имеет обратные зависимости между различными частями. Регистрация ошибок представляет собой особый случай управления событиями. Тревоги - это просто события, а сам регистратор ошибок - это контролируемый процесс, хотя процесс супервизора и может вызывать функции регистратора ошибок.

Я попытаюсь все это упорядочить и представить части в некотором осмысленном порядке. Итак, мы будем делать следующее:

Рассмотрим идеи использования типичного обработчика событий.

Увидим как работает регистратор ошибок.

Добавим управление тревогами.

Напишем два сервера.

Создадим дерево надзора и добавим в него наши серверы.

Упакуем всё в единое приложение.

18.1 Типичная обработка событий

Событие - это когда что-нибудь происходит, что-нибудь заслуживающее внимания программиста, который думает о том, кто и что должен делать в данном случае.

Когда мы программируем и происходит что-нибудь заметное, мы просто отправляем сообщение о событии зарегистрированному процессу. Что-то вроде этого:

RegProcName ! {event, E}

E - это событие (любой Эрланг-элемент (term)). RegProcName - имя зарегистрированного процесса.

Нам не нужно заботиться о том, что происходит с сообщением, когда мы его отправили. Мы просто выполнили работу и сообщили о том, что что-то случилось.

Теперь переключим наше внимание на процесс приёма сообщений о событиях. Этот процесс называется "обработчик событий". Простейший возможный обработчик событий - это "ничего не делающий" обработчик. Когда он принимает сообщение {event, X}, он ничего не делает с этим событием; просто отбрасывает его в сторону.

Вот наша первая попытка создания программы типичного обработчика событий:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"СкачатьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"eventHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"handlerHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/event_handler.erl"erl

```
-module(event_handler).

-export([make/1, add_handler/2, event/2]).

%% make a new event handler called Name

%% the handler function is noOp -- so we do nothing with the event

make(Name) ->

register(Name, spawn(fun() -> my_handler(fun no_op/1) end)).

add_handler(Name, Fun) -> Name ! {add, Fun}.

%% generate an event

event(Name, X) -> Name ! {event, X}.

my_handler(Fun) ->

receive

{add, Fun1} ->

my_handler(Fun1);
```

{event, Any} ->

(catch Fun(Any)),

my_handler(Fun)

end.

no_op(_) -> void.

API обработчика событий следующий:

event_handler:make(Name)

Приготовить "ничего не делающий" обработчик называемый Name (атом). Это то место, куда будут направляться события.

event_handler:event(Name, X)

Отправить событие X обработчику Name.

event_handler:add_handler(Name, Fun)

Добавить обработчик Fun к обработчику событий Name. Когда происходит событие X, обработчик выполнит Fun(X).

Теперь создадим обработчик и сгенерируем ошибку:

1> event_handler:make(errors).

true

2> event_handler:event(errors, hi).

{event,hi}

Ничего особенного не произойдет, потому что мы не подключили модуль обратных вызовов к этому обработчику.

Чтобы получить обработчик событий, который делает что-нибудь осмысленное, необходимо написать для него модуль обратных вызовов и подключить этот модуль к обработчику:

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"motorHYPERLINK

"http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"controllerHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/motor_controller.erl"erl

```
-module(motor_controller).

-export([add_event_handler/0]).

add_event_handler() ->

event_handler:add_handler(errors, fun controller/1).

controller(too_hot) ->

io:format("Turn off the motor\~n" );

controller(X) ->

io:format("\~w ignored event: \~p\~n" ,[?MODULE, X]).
```

Скомпилируем этот код и подключим к обработчику:

```
3> c(motor_controller).

{ok,motor_controller}

4> motor_controller:add_event_handler().

{add,#Fun<motor_controller.0.99476749>}
```

Теперь, когда события будут отправлены обработчику, они будут обработаны функцией motor_controller:controller/1:

```
5> event_handler:event(errors, cool).

motor_controller ignored event: cool

{event,cool}

6> event_handler:event(errors, too_hot).

Turn off the motor

{event,too_hot}
```

И в чём же смысл проделанной работы? Во первых, мы задали имя, на которое будут отправляться события. В данном случае, это зарегистрированный процесс errors.

Затем, мы определили протокол отправки событий зарегистрированному процессу. Но мы ничего не сказали о том, что происходит с событиями, которые получает этот процесс. На самом деле, всё что случается будет обработано в функции noOp(X). В конце мы подключим другой обработчик событий, но об этом позже.

Очень позднее связывание с "изменением ваших мыслей"

Предположим, что мы пишем функцию, которая скрывает конструкцию event_handler:event от программиста. Например, мы пишем следующее:

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

too_hot() ->

event_handler:event(errors, too_hot).

В этом случае мы говорим программисту вызывать lib_misc:too_hot() в своем коде, когда дела пойдут плохо. В большинстве языков программирования вызов функции too_hot был бы статически или динамически прилинкован в код программы. Так как вызов прилинкован, значит он выполняет фиксированную работу зависящую от кода. Если позднее изменится наше понимание и мы решим изменить что-нибудь, то это будет не простой путь изменения нашей системы.

Подход Эрланга к обработке событий абсолютно другой. Он позволяет отделить генерирацию событий от обработки событий. Мы можем изменить обработку в любое время, просто передав новую функцию обработки в обработчик событий. Ничего не линкуется статически, и каждый обработчик может быть изменен тогда, когда вам это потребуется.

Используя такой механизм, мы можем построить систему *меняющуюся со временем* и не требующую остановки для замены кода.

Примечание: Это не "позднее связывание" - это "ОЧЕНЬ позднее связывание, дающее возможность думать так или иначе".

Возможно, вы немного запутались. Почему мы говорим об обработчиках событий? Ключевой момент повествования в том, что обработчик событий предоставляет нам

инфраструктуру, в которую мы можем внедрять свои обработчики.

Инфраструктура регистратора ошибок строится из шаблона обработчика событий. Мы можем устанавливать различные обработчики в регистраторе ошибок для достижения различных целей.

18.2 Регистратор ошибок

Система OTP строится на настраиваемых регистраторах ошибок. Регистратор ошибок можно рассматривать с трех точек зрения. С точки зрения *программиста* - это вызовы функций позволяющие вести журнал ошибок. Точка зрения *конфигурации* - это то, как регистратор ошибок сохраняет данные. Точка зрения *отчетов* - это анализ ошибок после того как они случились. Мы рассмотрим каждую из точек зрения.

Журналирование/Протоколирование ошибок

Что касается программиста, API регистратора ошибок достаточно прост. Вот он:

@spec error_logger:error_msg(String) -> ok

Отправить сообщение об ошибке регистратору ошибок.

1> error_logger:error_msg("An error has occurred\n").

=ERROR REPORT==== 28-Mar-2007::10:46:28 ===

An error has occurred

ok

@spec error_logger:error_msg(Format, Data) -> ok

Отправить сообщение об ошибке регистратору ошибок. Аргументы такие же как и для io:format(Format, Data).

2> error_logger:error_msg("~s, an error has occurred\n", ["Joe"]).

=ERROR REPORT==== 28-Mar-2007::10:47:09 ===

Joe, an error has occurred

ok

@spec error_logger:error_report(Report) -> ok

Отправить стандартный отчет об ошибке регистратору ошибок.

• @type Report = [{Tag, Data} | term()] | string() | term()]

• @type Tag = term()

• @type Data = term()

3> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).

=ERROR REPORT==== 28-Mar-2007::10:51:51 ===

tag1: data1

a_term

tag2: data

Это только небольшая часть доступного API. Обсуждение деталей не очень интересно. В наших программах мы будем использовать только error_msg. Полное описание можно посмотреть на страницах руководства по error_logger.

Настройка регистратора ошибок

Существует много способов настроить регистратор ошибок. Мы можем видеть все ошибки в окне оболочки Эрланга (это режим по-умолчанию, специально настраивать не требуется). Мы можем записывать все ошибки попадающие в окно оболочки в один отформатированный файл. И, наконец, мы можем создать кольцевой ?циклический? журнал ошибок. Можете думать о колцевом журнале как о большом кольцевом буфере, который содержит сообщения, выдаваемые регистратором ошибок. Новые сообщения записываются в конец журнала, а когда журнал полон, то записи из начала журнала удаляются.

Кольцевые журналы используются очень часто. Вам решать как много файлов журналов использовать и насколько они будут большими, а система сама позаботится об удалении старых и создании новых файлов в кольцевом буфере. Вы можете задать подходящий размер файла, чтобы сохранить в нем записи за несколько дней, этого обычно достаточно в большинстве случаев.

Стандарные регистраторы ошибок

Когда мы запускаем Эрланг, мы можем использовать аргумент boot:

$ erl -boot start_clean

Такой запуск обеспечит окружение для разработки программ. Будет поддерживаться только простая регистрация ошибок. (Команда erl без аргумена boot эквивалентна команде erl -boot start_clean)

$ erl -boot start_sasl

Такой запуск обеспечит окружение для запуска системы готовой к эксплуатации. Библиотеки поддержки системной архитектуры (SASL - System Architecture Support Libraries) позаботится о регистрации ошибок, о перегрузках системы и так далее.

Настройку журналов лучше всего делать из файлов настроек, потому что вряд ли кто-нибудь помнит все аргументы регистратора. Далее мы рассмотрим как работает система по-умолчанию и увидим четыре конфигурации, которые меняют поведение регистратора.

SASL без настройки

Вот что происходит, когда мы запускаем SASL без файла настроек:

$ erl -boot start_sasl

Erlang (BEAM) emulator version 5.5.3 [async-threads:0] ...

=PROGRESS REPORT==== 27-Mar-2007::11:49:12 ===

supervisor: {local,sasl_safe_sup}

started: [{pid,<0.32.0>},

{name,alarm_handler},

{mfa,{alarm_handler,start_link,[]}},

{restart_type,permanent},

{shutdown,2000},

{child_type,worker}]

... many lines removed ...

Eshell V5.5.3 (abort with \^G)

Сейчас мы вызовем одну из конструкций error_logger для отчета об ошибке:

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error

ok

Заметим, что отчет об ошибке отобразился в оболочке Эрланга. Вывод отчетов об ошибках зависит от настроек регистратора ошибок.

Управление регистратором

Регистратор ошибок предоставляет несколько типов отчетов:

Отчеты супервизора

Отчеты о том, что OTP супервизор запускает или останавливает подчинённые процессы (мы поговорим о супервизорах а разделе 18.5 "Дерево надзора", на странице 351).

Отчеты о выполнении

Отчеты о запуске или остановке супрвизора.

Отчеты об авариях

Отчеты об остановке выполнения с сообщением о причине отказа помимо normal или shutdown.

Эти три типа отчетов обрабатываются автоматически и не требуют вмешательства программиста.

Мы можем дополнительно вызвать конкретную конструкцию модуля error_handler, чтобы обработать все три типа отчетов. Это позволит нам использовать соощения об ошибках, предупреждения и сообщения информационного характера. Три этих термина ничего не означают; воспринимайте их как теги, позволяющие программисту различать природу элементов в журнале ошибок.

Позже, когда журнал ошибок будет проанализирован, эти теги помогут нам решить какие из элементов журнала исследовать. Когда мы настраиваем регистратор ошибок, мы можем указать, что требуется сохранять только ошибки, а все остальные элементы игнорировать. Теперь давайте напишем файл настроек elog1.config для настройки регистратора ошибок:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"1.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog1.config"config

```
%% no tty
```

[{sasl, [

{sasl_error_logger, false}

]}].

Если мы запустим систему с этим файлом настроек, то будем получать только сообщения об ошибках, мы не получим сообщений о ходе выполнения и прочих. Все эти сообщения будут выводиться только в окно оболочки Эрланга.

$ erl -boot start_sasl -config elog1

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error

ok

Текстовый файл и оболочка (shell)

Следующий файл настроек выдаёт список ошибок в окно оболочки Эрланга и дублирует все сообщения из оболочки в файл:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"2.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog2.config"config

%% single text file - minimal tty

[{sasl, [

%% All reports go to this file

{sasl_error_logger, {file, "/home/joe/error_logs/THELOG" }}

]}].

Для проверки мы запустим Эрланг, сгенерируем сообщение об ошибке и посмотрим результат в файле:

$ erl -boot start_sasl -config elog2

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 27-Mar-2007::11:53:08 ===

This is an error ok

Если мы посмотрим файл /home/joe/error_logs/THELOG, в начале файла мы найдем следующие строки:

=PROGRESS REPORT==== 28-Mar-2007::11:30:55 ===

supervisor: {local,sasl_safe_sup}

started: [{pid,<0.34.0>},

{name,alarm_handler},

{mfa,{alarm_handler,start_link,[]}},

{restart_type,permanent},

{shutdown,2000},

{child_type,worker}]

...

Кольцевой журнал и оболочка

Эта конфигурация даст нам возможность выводить все ошбики в оболочку Эрланга плюс дублирование всего вывода оболочки в кольцевой журнальный файл. Такой вариант наиболее часто используется на практике.

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"elogHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"3.HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/elog3.config"config

%% rotating log and minimal tty

[{sasl, [

{sasl_error_logger, false},

%% задать параметры кольцевого журнала

%% директория с файлом журнала

{error_logger_mf_dir,"/home/joe/error_logs" },

%% # кол-во байт выделенное для журнала

{error_logger_mf_maxbytes,10485760}, % 10 MB

%% максимальное кол-во файлов-журналов

{error_logger_mf_maxfiles, 10}

]}].

$erl -boot start_sasl -config elog3

1> error_logger:error_msg("This is an error\n").

=ERROR REPORT==== 28-Mar-2007::11:36:19 ===

This is an error

false

При запуске системы все ошибки будут направляться в файл кольцевого журнала. Позже в этой главе мы рассмотрим как извлекать эти ошибки из файла-журнала.

Продуктивная среда ?!ПРОДАКШЕН!?

В продуктивной среде нам, на самом деле, интересны только отчеты об ошибках, а не о процессе выполнения или какая-либо информация, поэтому заставим регистратор отчитываться только об ошибках. Без этих настроек систему будут перегружать информационные отчеты и отчеты о процессе исполнения.

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"elogHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"4.HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/elog4.config"config

%% rotating log and errors

[{sasl, [

%% minimise shell error logging

{sasl_error_logger, false},

%% only report errors

{errlog_type, error},

%% define the parameters of the rotating log

%% the log file directory

{error_logger_mf_dir,"/home/joe/error_logs" },

%% # bytes per logfile

{error_logger_mf_maxbytes,10485760}, % 10 MB

%% maximum number of

{error_logger_mf_maxfiles, 10}

]}].

В результате запуска получим нечто похожее на предыдущий пример. С той лишь разницей, что регистрироваться будут только ошибки.

Анализируем ошибки

Чтение журнала ошибок входит в обязанности модуля rb. Этот модуль имеет чрезвычайно простой интерфейс.

1> rb:help().

Report Browser Tool - usage

---

rb:start() - start the rb_server with default options

rb:start(Options) - where Options is a list of:

{start_log, FileName}

- default: standard_io

{max, MaxNoOfReports}

- MaxNoOfReports should be an integer or 'all'
- default: all

...

... many lines omitted ...

...

Запустим браузер отчетов и скажем ему сколько записей из журнала читать (в данном случае последние двадцать):

2> rb:start([{max,20}]).

rb: reading report...done.

3> rb:list().

No Type Process Date Time

== ==== ======= ==== ====

11 progress <0.29.0> 2007-03-28 11:34:31

10 progress <0.29.0> 2007-03-28 11:34:31

9 progress <0.29.0> 2007-03-28 11:34:31

8 progress <0.29.0> 2007-03-28 11:34:31

7 progress <0.22.0> 2007-03-28 11:34:31

6 progress <0.29.0> 2007-03-28 11:35:53

5 progress <0.29.0> 2007-03-28 11:35:53

4 progress <0.29.0> 2007-03-28 11:35:53

3 progress <0.29.0> 2007-03-28 11:35:53

2 progress <0.22.0> 2007-03-28 11:35:53

1 error <0.23.0> 2007-03-28 11:36:19

ok

> rb:show(1).

ERROR REPORT <0.40.0> 2007-03-28 11:36:19

This is an error

ok

Для того чтобы найти конкретную ошибку мы можем использовать такую команду как rb:grep(RegExp), при её использовании найдется то, что описано в регулярном выражении RegExp. Я не хочу углубляться в то, как анализировать журналы ошибок. Лучше потратьте время и поинтересуйтесь модулем rb и все увидите сами. Замечу, что на самом деле вам никогда не потребуется удалять журналы ошибок, точный механизм кольцевых журналов в конце концов сам удалит старые записи.

Если вам требуется оставить все сообщения об ошибках, вы можете задать интервалы и удалять информацию по мере необходимости.

18.3 Управление тревогами

Когда мы пишем наше приложение, нам требуется только одна тревога - будем реагировать только тогда, когда начнет перегреваться процессор, потому что мы вычисляем гигантские простые числа (помните, мы создали компанию по продаже простых чисел). Вот теперь-то мы и будем использовать настоящий OTP-шный обработчик тревог (и не простой как в начале главы).

Обработчик тревог это модуль обратных вызовов OTP для поведения gen_event. Вот его код:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"ЗагрузатьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"myHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"*HYPERLINK*
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"alarmHYPERLINK
*"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"*HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"handlerHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/my_alarm_handler.erl"erl

```
-module(my_alarm_handler).

-behaviour(gen_event).

%% gen_event callbacks

-export([init/1, handle_event/2, handle_call/2,

handle_info/2, terminate/2]).
```

```
%% init(Args) must return {ok, State}

init(Args) ->

io:format("*** my_alarm_handler init:\~p\~n" ,[Args]),

{ok, 0}.

handle_event({set_alarm, tooHot}, N) ->

error_logger:error_msg("*** Tell the Engineer to turn on the fan\~n" ),

{ok, N+1};

handle_event({clear_alarm, tooHot}, N) ->

error_logger:error_msg("*** Danger over. Turn off the fan\~n" ),

{ok, N};

handle_event(Event, N) ->

io:format("*** unmatched event:\~p\~n" ,[Event]),

{ok, N}.

handle_call(_Request, N) -> Reply = N, {ok, N, N}.

handle_info(_Info, N) -> {ok, N}.

terminate(Reason, N) -> ok.
```

Этот код очень похож на код обратных вызовов gen_server, который мы видели раньше в разделе 16.3, "*Что же происходит когда мы вызываем сервер*?", на странице 306. Интересующей нас конструкцией является handle_event(Event, State). Она возвращает {ok, NewState}. Event - это кортеж имеющий форму {EventType, EventArg}, где EventType это атом set_event или clear_event, а EventArg - это пользовательские аргументы. Чуть позже мы рассмотрим как генерируются такие события.

А теперь позабавимся. Мы запустим систему, сгенерируем тревогу, установим обработчик тревог, сгенерируем новую тревогу, и так далее:

```
$ erl -boot start_sasl -config elog3

1> alarm_handler:set_alarm(tooHot).

ok
```

=INFO REPORT==== 28-Mar-2007::14:20:06 ===

alarm_handler: {set,tooHot}

2> gen_event:swap_handler(alarm_handler,

{alarm_handler, swap},

{my_alarm_handler, xyz}).

*** my_alarm_handler init:{xyz,{alarm_handler,[tooHot]}}

3> alarm_handler:set_alarm(tooHot).

ok

=ERROR REPORT==== 28-Mar-2007::14:22:19 ===

*** Tell the Engineer to turn on the fan

4> alarm_handler:clear_alarm(tooHot).

ok

=ERROR REPORT==== 28-Mar-2007::14:22:39 ===

*** Danger over. Turn off the fan

Что же здесь присходит?

Мы запустили Эрланг с -boot start_sasl. Когда мы сделали это, мы получили стандартный обработчик тревог. Когда мы устанавливаем или очищаем тревогу, то ничего не происходит. Это простой "ничего не делающий" обработчик событий, мы такие рассматривали раньше.

Когда мы установили тревогу (строка 1), мы просто получили информационный отчет. Здесь нет специальной обработки тревог.

Мы установили свой обработчик тревог (строка 2). Аргумент в my_alarm_handler (xyz) не имеет особого значения; синтаксис требует какое-нибудь значение, но поскольку нам не требуются значения, мы просто используем атом xyz, мы сможем увидеть этот аргумент при выводе на консоль.

Строка •• my_alarm_handler_init: ... напечатана из нашего модуля обратных вызовов.

Мы установили и очистили тревогу tooHot (строки 3 и 4). Это отработал наш обработчик тревог. Мы можем проверить, прочитав вывод на консоли.

Чтение журнала

Давайте вернёмся обратно к регистратору ошибок и посмотрим, что там происходит:

```
1> rb:start([{max,20}]).

rb: reading report...done.

2> rb:list().

No Type Process Date Time

== ==== ======= ==== ====

...

3 info_report <0.29.0> 2007-03-28 14:20:06

2 error <0.29.0> 2007-03-28 14:22:19

1 error <0.29.0> 2007-03-28 14:22:39

3> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 14:22:39

---

*** Danger over. Turn off the fan

ok

4> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 14:22:19

---

*** Tell the Engineer to turn on the fan
```

Итак, здесь мы видим как работает механизм регистратора ошибок.

На практике мы должны были бы убедиться, что журнал ошибок достаточно велик для хранения данных за несколько дней или даже недель. Каждые несколько дней (или недель) мы бы проверяли журнал на предмет ошибок.

Примечание: Модуль rb содержит функции для выбора ошибок указанного типа и извлечения этих ошибок в файл. В результате процесс анализа ошибок может быть полностью автоматизирован.

18.4 Серверные приложения

Наше приложение состоит из двух серверов: сервер простых чисел и сервер рассчёта площади. Рассмотрим сервер простых чисел. Он написан с использованием поведения gen_server (см. раздел 16.2 "*Начинаем с gen_server*" на стр. 301). Замечу, что он включает в себя обработку тревог которую мы разработали в предыдущем разделе.

Сервер простых чисел

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"primeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"serverHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"erl

```
-module(prime_server).

-behaviour(gen_server).

-export([new_prime/1, start_link/0]).

%% gen_server callbacks

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,

terminate/2, code_change/3]).

start_link() ->

gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

new_prime(N) ->

%% 20000 is a timeout (ms)

gen_server:call(?MODULE, {prime, N}, 20000).

init([]) ->

%% Note we must set trap_exit = true if we

%% want terminate/2 to be called when the application

%% is stopped
```

```
process_flag(trap_exit, true),
io:format("\~p starting\~n" ,[?MODULE]),
{ok, 0}.
handle_call({prime, K}, _From, N) ->
{reply, make_new_prime(K), N+1}.
handle_cast(_Msg, N) -> {noreply, N}.
handle_info(_Info, N) -> {noreply, N}.
terminate(Reason, N) ->
io:format("\~p stopping\~n" ,[?MODULE]),
ok.
code_change(OldVsn, N, Extra) -> {ok, N}.
make_new_prime(K) ->
if
K > 100 ->
alarm_handler:set_alarm(tooHot),
N = lib_primes:make_prime(K),
alarm_handler:clear_alarm(tooHot),
N;
true ->
lib_primes:make_prime(K)
end.
```

Сервер площади

Теперь рассмотрим сервер площади. Он так же построен на поведении gen_server. Заметьте, написание сервера очень быстрый процесс. Когда я писал этот пример, я просто скопировал код из сервера простых чисел и вставил его в новый сервер. Все заняло несколько минут.

Сервер площади не является идеальной программой и содержит преднамеренную ошибку (сможете ее найти?). Мой не очень коварный план - это заставить сервер рухнуть, чтобы быть перестрахованным супервизором. А потом получить отчёт обо всех ошибках в журнале ошибок.

HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"ЗагрузитьHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl" HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"areaHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"_HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"serverHYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl".HYPERLINK "http://media.pragprog.com/titles/jaerlang/code/prime_server.erl"erl

```
-module(area_server).

-behaviour(gen_server).

-export([area/1, start_link/0]).

%% gen_server callbacks

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,

terminate/2, code_change/3]).

start_link() ->

gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

area(Thing) ->

gen_server:call(?MODULE, {area, Thing}).

init([]) ->

%% Note we must set trap_exit = true if we

%% want terminate/2 to be called when the application

%% is stopped

process_flag(trap_exit, true),

io:format("\~p starting\~n" ,[?MODULE]),

{ok, 0}.
```

handle_call({area, Thing}, _From, N) -> {reply, compute_area(Thing), N+1}.

handle_cast(_Msg, N) -> {noreply, N}.

handle_info(_Info, N) -> {noreply, N}.

terminate(*Reason,* N) ->

io:format("\~p stopping\~n" ,[?MODULE]),

ok.

code_change(*OldVsn, N,* Extra) -> {ok, N}.

compute_area({square, X}) -> X*X;

compute_area({rectonge, X, Y}) -> X*Y.

надзор one_for_one

Если один процесс рухнет, он будет перезапущен

надзор all_for_one

Если один процесс рухнет, все процессы будут прерваны и перезапущены

18.1 Два вида дерева надзора

18.5 Дерево надзора

Дерево надзора - это дерево процессов. Самые верхние процессы (супервизоры) в дереве наблюдают за нижними (рабочими) процессами в дереве и перезапускют нижние процессы, если те аварийно завершаются. Два вида дерева надзора вы можете увидеть на рисунке 18.1.

One-for-one дерево надзора

В надзоре one-for-one, если один процесс рухнул, то супервизор рестартует

только этот процесс.

All-for-one дерево надзора

В надзоре all-for-one, если любой из процессов рухнет, то все поднадзорные

процессы будут уничтожены (вызовом функции terminate/2 в соответстующем

модуле обратных вызовов). Затем все рабочие процессы будут рестартованы.

Супервизоры создаются с использованием OTP поведения *supervisor*. Это поведение описывается в специальном модуле обратных вызовов, который содержит стратегию надзора и правила запуска отдельных рабочих процессов в дереве надзора. Дерево надзора определяется функцией следующего вида:

init(...) ->

{ok, {RestartStrategy, MaxRestarts, Time},

[Worker1, Worker2, ...]}.

Здесь RestartStrategy это один из атомов one_for_one или all_for_one. MaxRestarts и Time указывают на "частоту перезапуска". Если супервизор перезапускает процессы большее число раз, чем указано в MaxRestarts за Time секунд, то работа супервизора будет прервана. Это делается для того, чтобы остановить бесконечный цикл перезапуска процессов, если они содержат ошибки и останавливаются из-за них.

Worker1, Worker2 и т.д. это кортеж описывающий как запускать каждый из рабочих процессов. Мы увидим, как это выглядит уже скоро.

Теперь давайте вернемся к нашей компании и создадим дерево надзора.

Для начала, думаю, нам надо выбрать имя для нашей компании. Пусть будет sellaprime. Задача супервизора sellaprime - это конечно же держать всегда запущенными сервер простых чисел и сервер площади. Для этого напишем уже другой модуль обратных вызовов, теперь для gen_supervisor. Вот этот модуль:

HYPERLINK ""ЗагрузитьHYPERLINK "" HYPERLINK ""sellaprimeHYPERLINK ""_HYPERLINK ""supervisorHYPERLINK "".HYPERLINK ""erl

-module(sellaprime_supervisor).

-behaviour(supervisor). % see erl -man supervisor

-export([start/0, start_in_shell_for_testing/0, start_link/1, init/1]).

start() ->

spawn(fun() ->

supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = [])

end).

start_in_shell_for_testing() ->

{ok, Pid} = supervisor:start_link({local,?MODULE}, ?MODULE, _Arg = []),

```
unlink(Pid).

start_link(Args) ->

supervisor:start_link({local,?MODULE}, ?MODULE, Args).

init([]) ->

%% Install my personal error handler

gen_event:swap_handler(alarm_handler,

{alarm_handler, swap},

{my_alarm_handler, xyz}),

{ok, {{one_for_one, 3, 10},

[{tag1,

{area_server, start_link, []},

permanent,

10000,

worker,

[area_server]},

{tag2,

{prime_server, start_link, []},

permanent,

10000,

worker,

[prime_server]}

]}}.
```

Самая важная часть - это структура данных возвращаемая функцией init/1:

HYPERLINK ""ЗагрузитьHYPERLINK "" HYPERLINK ""sellaprimeHYPERLINK ""_HYPERLINK ""supervisorHYPERLINK "".HYPERLINK ""erl

{ok, {{one_for_one, 3, 10},

[{tag1,

{area_server, start_link, []},

permanent,

10000,

worker,

[area_server]},

{tag2,

{prime_server, start_link, []},

permanent,

10000,

worker,

[prime_server]}

]}}.

Эта структура данных определяет стратегию надзора. Мы говорили о стратегии надзора и частоте перезапуска выше. Сейчас осталось дать определение для сервера областей и сервера простых чисел.

Определение для Worker процессов имеет следующий вид:

{Tag, {Mod, Func, ArgList},

Restart,

Shutdown,

Type,

[Mod1]}

Что же обозначают все эти аргументы?

Tag

Атом, который будет использоваться для ссылки на рабочий процесс в дальнейшем

(если потребуется).

{Mod, Func, ArgList}

Определение функции, которую супервизор будет использовать для запуска рабочего процесса. Оно используется как аргумент при вызове apply(Mod, Fun, ArgList).

Restart = permanent | transient | temporary

permanent - процесс будет перезапускаться всегда. transient - процесс будет перезапущен только, если получено ненормальное значение при выходе. temporary - процесс запускается только один раз и не перезапускается.

Shutdown

Время остановки. Это максимально разрешенное время для остановки рабочего процесса. Если время остановки процесса будет превышено, то процесс просто будет убит. (Возможны и другие значения - см. руководство по Супервизору)

Type = worker | supervisor

Тип надзираемого процесса. Мы можем сконструировать дерево надзора над супервизорами, добавляя процесс супервизора вместо рабочего процесса.

[Mod1]

Это имя модуля обратных вызовов, если дочерний процесс имеет поведение supervisor или gen_server (Возможны и другие значения - см. руководство по Супервизору)

18.6 Запуск системы

Теперь мы готовы первый раз запустить нашу компанию. Мы вернулись. Кто хочет купить первое простое число?

Давайте запустим систему:

$ erl -boot start_sasl -config elog3

1> sellaprime_supervisor:start_in_shell_for_testing().

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

Теперь сделаем правильный запрос:

2> area_server:area({square,10}).

100

Сейчас сделаем неправильный запрос:

3> area_server:area({rectangle,10,20}).

area_server stopping

=ERROR REPORT==== 28-Mar-2007::15:15:54 ===

** Generic server area_server terminating

** Last message in was {area,{rectangle,10,20}}

Действительно ли работает стратегия надзора?

Эрланг был разработан для программирования отказоустойчивых систем. Первоначальная разработка была сделана в Лаборатории Вычислительной Техники Шведской компании Эрикссон. С тех пор группа OTP вела разработку с помощью десятков сотрудников компании. Используя gen_server, gen_supervisor и другие поведения Эрланга строились системы с надежностью 99.9999999% (тут девять девяток). При правильном использовании, механизм обработки ошибок может помочь сделать вашу программу работающей вечно (ну, или почти вечно). Регистратор ошибок, описанный здесь, работает уже в течение нескольких лет в реальных, живых продуктах.

** When Server state == 1

** Reason for termination ==

** {function_clause,[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]}

area_server starting

** exited: {{function_clause,

[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]},

{gen_server,call,

[area_server,{area,{rectangle,10,20}}]}} **

Упс - что же тут случилось? Сервер площади рухнул; мы умышленно допустили в коде ошибку. Авария была обнаружена супервизором и сервер был перезапущен. Все это запротоколировал регистратор ошибок.

После аварии, все вернулось к нормальному состоянию, как и должно было. Давайте сейчас сделаем правильный запрос:

4> area_server:area({square,25}).

625

У нас все опять работает. Теперь давайте сгенерируем маленькое простое число:

5> prime_server:new_prime(20).

Generating a 20 digit prime .....

37864328602551726491

А теперь сгенерируем большое простое число:

6> prime_server:new_prime(120).

Generating a 120 digit prime

=ERROR REPORT==== 28-Mar-2007::15:22:17 ===

*** Tell the Engineer to turn on the fan

.....

=ERROR REPORT==== 28-Mar-2007::15:22:20 ===

*** Danger over. Turn off the fan

765525474077993399589034417231006593110007130279318737419683

288059079481951097205184294443332300308877493399942800723107

Теперь у нас работоспособная система. Если на сервере случится авария, то он автоматически будет перезапущен, а регистратор ошибок проинформирует нас об

этом.

Сейчас давайте рассмотрим журнал ошибок:

```
1> rb:start([{max,20}]).
rb: reading report...done.
rb: reading report...done.
{ok,<0.53.0>}
2> rb:list().
No Type Process Date Time
== ==== ======= ==== ====
20 progress <0.29.0> 2007-03-28 15:05:15
19 progress <0.22.0> 2007-03-28 15:05:15
18 progress <0.23.0> 2007-03-28 15:05:21
17 supervisor_report <0.23.0> 2007-03-28 15:05:21
16 error <0.23.0> 2007-03-28 15:07:07
15 error <0.23.0> 2007-03-28 15:07:23
14 error <0.23.0> 2007-03-28 15:07:41
13 progress <0.29.0> 2007-03-28 15:15:07
12 progress <0.29.0> 2007-03-28 15:15:07
11 progress <0.29.0> 2007-03-28 15:15:07
10 progress <0.29.0> 2007-03-28 15:15:07
9 progress <0.22.0> 2007-03-28 15:15:07
8 progress <0.23.0> 2007-03-28 15:15:13
7 progress <0.23.0> 2007-03-28 15:15:13
6 error <0.23.0> 2007-03-28 15:15:54
5 crash_report area_server 2007-03-28 15:15:54
```

4 supervisor_report <0.23.0> 2007-03-28 15:15:54

3 progress <0.23.0> 2007-03-28 15:15:54

2 error <0.29.0> 2007-03-28 15:22:17

1 error <0.29.0> 2007-03-28 15:22:20

Что-то тут не так. У нас есть отчет об аварии сервера областей. Как узнать, что случилось (если бы мы не знали об этом)?

9> rb:show(5).

CRASH REPORT <0.43.0> 2007-03-28 15:15:54

Crashing process

pid <0.43.0>

registered_name area_server

error_info

{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]}

initial_call

{gen,init_it,

[gen_server,

<0.42.0>,

<0.42.0>,

{local,area_server},

area_server,

[],

[]]}

ancestors [sellaprime_supervisor,<0.40.0>]

messages []

links [<0.42.0>]

dictionary []

trap_exit false

status running

heap_size 233

stack_size 21

reductions 199

ok

Распечатка {function_clause, compute_area, ...} отображает точное место в программе сервера где произошла авария. Это должно помочь легко локализовать и исправить ошибку. Давайте перейдем к рассмотрению следующих ошибок:

10> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 15:22:17

*** Tell the Engineer to turn on the fan

и

10> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 15:22:20

*** Danger over. Turn off the fan

Это предупреждения нашей системы охлаждения при вычислении очень больших простых чисел!

18.7 Приложение

Мы почти закончили. Всё что нам осталось сделать - это написать файл с расширением .app который будет содержать информацию о нашем приложении:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"sellaprimeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"app

```
%% This is the application resource file (.app file) for the 'base'

%% application.

{application, sellaprime,

[{description, "The Prime Number Shop" },

{vsn, "1.0" },

{modules, [sellaprime_app, sellaprime_supervisor, area_server,

prime_server, lib_primes, my_alarm_handler]},

{registered,[area_server, prime_server, sellaprime_super]},

{applications, [kernel,stdlib]},

{mod, {sellaprime_app,[]}},

{start_phases, []}

]}.
```

Теперь нам потребуется написать модуль обратных вызовов с именем модуля из предыдущего примера:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"sellaprimeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"appHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"erl

```
-module(sellaprime_app).

-behaviour(application).
```

```
-export([start/2, stop/1]).

%%--------------------------------------------------------------------

%% Function: start(Type, StartArgs) -> {ok, Pid} |

%% {ok, Pid, State} |

%% {error, Reason}

%% Description: This function is called whenever an application

%% is started using application:start/1,2, and should start the processes

%% of the application. If the application is structured according to the

%% OTP design principles as a supervision tree, this means starting the

%% top supervisor of the tree.

%%--------------------------------------------------------------------

start(_Type, StartArgs) ->

sellaprime_supervisor:start_link(StartArgs).

%%--------------------------------------------------------------------

%% Function: stop(State) -> void()

%% Description: This function is called whenever an application

%% has stopped. It is intended to be the opposite of Module:start/2 and

%% should do any necessary cleaning up. The return value is ignored.

%%--------------------------------------------------------------------

stop(_State) ->

ok.
```

Здесь должны быть экспортированы функции start/2 и stop/1. Раз мы уже все это сделали, то теперь можем запустить наше приложение в оболочке Эрланга.

```
$ erl -boot start_sasl -config elog3

1> application:loaded_applications().
```

```
[{kernel,"ERTS CXC 138 10","2.11.3"},

{stdlib,"ERTS CXC 138 10","1.14.3"},

{sasl,"SASL CXC 138 11","2.1.4"}]

2> application:load(sellaprime).

ok

3> application:loaded_applications().

[{sellaprime,"The Prime Number Shop","1.0"},

{kernel,"ERTS CXC 138 10","2.11.3"},

{stdlib,"ERTS CXC 138 10","1.14.3"},

{sasl,"SASL CXC 138 11","2.1.4"}]

4> application:start(sellaprime).

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

ok

5> application:stop(sellaprime).

prime_server stopping

area_server stopping

=INFO REPORT==== 2-Apr-2007::19:34:44 ===

application: sellaprime

exited: stopped

type: temporary

ok

6> application:unload(sellaprime).

ok
```

7> application:loaded_applications().

[{kernel,"ERTS CXC 138 10","2.11.4"},

{stdlib,"ERTS CXC 138 10","1.14.4"},

{sasl,"SASL CXC 138 11","2.1.5"}]

Вот теперь это вполне оперившееся приложение. Во второй строке мы загрузили приложение; этот вызов загружает весь код, но не запускает приложение. В четвертой строке мы запустили приложение, а в пятой строке остановили его. Заметьте, все видно в распечатке, когда приложения запускаются и останавливаются, соответствующие функции сервера простых чисел и сервера площади были вызваны. В шестой строке мы выгрузили приложение. Все модули приложения были удалены из памяти.

Когда мы делаем полноценную систему, используя OTP, мы упаковываем её в приложение. Это даёт нам универсальный метод запуска, остановки и управления приложением.

Заметьте, когда мы используем init:stop() для завершения работы системы, то все приложения будут завершены так, как это принято. Это правило хорошего тона.

$ erl -boot start_sasl -config elog3

1> application:start(sellaprime).

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

ok

2> init:stop().

ok

prime_server stopping

area_server stopping

$

Две строки следующие за командой номер 2 получены из сервера площади и сервера простых чисел, они показывают нам, что был вызван метод terminate/2 из модуля

обратных вызовов gen_server.

18.8 Организация файловой системы

Я до сих пор ничего не упоминал об организации файловой системы. Это сделано специально - моя цель озадачивать вас проблемами по одной.

Структурированное OTP приложение, обычно, содержит файлы соответствующие различным частям приложения в строго определенных местах. Это не требование; так как все нужные файлы могут быть найдены во время исполнения, но это не потребуется, если все файлы лежат в нужных местах.

Все описанные в этой книге демонстрационные файлы располагаются в одной директории. Это простейшие примеры, и сделано это во избежание проблем с путями поиска и с взаимодействием между различными програмами.

Основные файлы, используемые в компании sellaprime, следующие:

File

Content

area_server.erl

Сервер областей - модуль обратных вызовов gen_server

prime_server.erl

Сервер простых чисел - модуль обратных вызовов gen_server

sellaprim_supervisor.erl

Модуль обратных вызовов Супервизора

sellaprim_app.erl

Модуль обратных вызовов Приложения

my_alam_handler.erl

Модуль обратных вызовов Событий для gen_event

sellaprime.app

Спецификация приложения

elog4.config

Файл настроек Регистратора ошибок

Для рассмотрения того как используются эти файлы и модули нам нужно рассмотреть последовательность событий, происходящих, когда приложение стартует:

Мы запускаем систему следующими командами: $ erl -boot start_sasl -config elog4.config 1> application:start(sellaprime). ...

Файл sellaprime.app должен находиться в корневом каталоге из которого запускается Эрланг или в подкаталоге этого каталога.

В этом случае контроллер приложения сможет обнаружить {mod, ...}, объявленый в sellaprime.app. Он содержит имя контроллера приложения. И конечно же это модуль sellaprime_app.

Вызвается обратный вызов sellaprime_app:start/2.

sellaprime_app:start/2 вызывает sellaprime_supervisor:start_link/2, который, в свою очередь, запускает супервизор sellaprime.

Вызвается обратный вызов супервизора sellaprime_supervisor:init/1 - он устанавливает обработчик ошибок и возвращает спецификацию надзора. Спецификация надзора сообщает как запускать сервер площади и сервер простых чисел.

sellaprime супервизор запускает сервер площади и сервер простых чисел. Они реализованы как модули обратных вызовов gen_server.

Останавливать все это очень просто. Вы просто вызываете application:stop(sellaprime) или init:stop().

18.9 Монитор приложений

Монитор приложений - это программа с графическим интерфейсом (GUI) для просмотра запущенных приложений. Команда appmon:start() запускает монитор приложений. Когда вы выполните эту команду, вы увидите окно, похожее на то, которое изображено на рисунке 18.2. Для того чтобы увидеть структуру приложения, вы должны кликнуть по одному из приложений. Монитор приложений для приложения sellaprime показан на рисунке 18.3.

18.10 Копаем глубже

Я пропустил довольно много подробностей, разъяснив только принципы. Вы сможете найти подробности на страницах руководства по gen_event, error_logger, supervisor и application.

18.2 Монитор приложений. Вид при запуске.

Рисунок 18.3 Приложение sellaprime

Следующие файлы содержат более подробную информацию о том как использовать OTP поведения:

HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"httpHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"://HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"wwwHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"erlangHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"orgHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"docHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"pdfHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"designHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"_HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"principlesHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"pdf

(страница 97) Gen servers, gen event, supervisors

HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"httpHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"://HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"wwwHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"erlangHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"orgHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"docHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"pdfHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"systemHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"_HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"principlesHYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK

"http://www.erlang.org/doc/pdf/system_principles.pdf"pdf

(страница 19) Как сделать boot файл

HYPERLINK "http://www.erlang.org/doc/pdf/appmon.pdf"httpHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"://HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"wwwHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"erlangHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"orgHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"docHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"pdfHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"appmonHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"pdf

(страница 16) Монитор приложений

18.11 Как мы вычисляем простые числа?

Очень просто.

```
%% make a prime with at least K decimal digits.

%% Here we use 'Bertrand's postulate.

%% Bertrands postulate is that for every N > 3,

%% there is a prime P satisfying N < P < 2N - 2

%% This was proved by Tchebychef in 1850

%% (Erdos improved this proof in 1932)

make_prime(1) ->

lists:nth(random:uniform(5), [1,2,3,5,7]);

make_prime(K) when K > 0 ->

new_seed(),
```

```
N = make_random_int(K),
if N > 3 ->
io:format("Generating a \~w digit prime " ,[K]),
MaxTries = N - 3,
P1 = make_prime(MaxTries, N+1),
io:format("\~n" ,[]),
P1;
true ->
make_prime(K)
end.
make_prime(0, _) ->
exit(impossible);
make_prime(K, P) ->
io:format("." ,[]),
case is_prime(P) of
true -> P;
false -> make_prime(K-1, P+1)
end.
%% Fermat's little theorem says that if
%% N is a prime and if A < N then
%% A\^N mod N = A
is_prime(D) ->
new_seed(),
is_prime(D, 100).
is_prime(D, Ntests) ->
```

```
N = length(integer_to_list(D)) -1,
is_prime(Ntests, D, N).
is_prime(0, , ) -> true;
is_prime(Ntest, N, Len) ->
K = random:uniform(Len),
%% A is a random number less than N
A = make_random_int(K),
if
A < N ->
case lib_lin:pow(A,N,N) of
A -> is_prime(Ntest-1,N,Len);
_ -> false
end;
true ->
is_prime(Ntest, N, Len)
end.
1> lib_primes:make_prime(500).
Generating a 500 digit prime .....
791015726987201027909055597115096126908592921342508297266243912592631402855283461324397013307924771094786030944973946964404399696758714374940531222422946966707622926139385002096578309062534166780603261012226023459181325555764028306928844115181391107802007557066746476035515105154017421267382367314941956505578474497545252666718280976890401503018406521440650857349061213980678938094352667372672691906693169783133618111423622890421398067893809435266737267269190669316978313361811142362289042139806789380943526673726726919066931697831336181114236228904
```

01868042872198074546193740053777668271056036892838181730070340

056505784153

red_name area_server

error_info

{function_clause,[{area_server,compute_area,[{rectangle,10,20}]},

{area_server,handle_call,3},

{gen_server,handle_msg,6},

{proc_lib,init_p,5}]}

initial_call

{gen,init_it,

[gen_server,

<0.42.0>,

<0.42.0>,

{local,area_server},

area_server,

[],

[]]}

ancestors [sellaprime_supervisor,<0.40.0>]

messages []

links [<0.42.0>]

dictionary []

trap_exit false

status running

heap_size 233

stack_size 21

reductions 199

ok

Распечатка {function_clause, compute_area, ...} отображает точное место в программе сервера где произошла авария. Это должно помочь легко локализовать и исправить ошибку. Давайте перейдем к рассмотрению следующих ошибок:

10> rb:show(2).

ERROR REPORT <0.33.0> 2007-03-28 15:22:17

*** Tell the Engineer to turn on the fan

и

10> rb:show(1).

ERROR REPORT <0.33.0> 2007-03-28 15:22:20

*** Danger over. Turn off the fan

Это предупреждения нашей системы охлаждения при вычислении очень больших простых чисел!

18.7 Приложение

Мы почти закончили. Всё что нам осталось сделать - это написать файл с расширением .app который будет содержать информацию о нашем приложении:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"sellaprimeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime.app"app

%% This is the application resource file (.app file) for the 'base'

%% application.

{application, sellaprime,

[{description, "The Prime Number Shop" },

{vsn, "1.0" },

{modules, [sellaprime_app, sellaprime_supervisor, area_server,

prime_server, lib_primes, my_alarm_handler]},

{registered,[area_server, prime_server, sellaprime_super]},

{applications, [kernel,stdlib]},

{mod, {sellaprime_app,[]}},

{start_phases, []}

]}.

Теперь нам потребуется написать модуль обратных вызовов с именем модуля из предыдущего примера:

HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"ЗагрузитьHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl" HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"sellaprimeHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"_HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"appHYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl".HYPERLINK
"http://media.pragprog.com/titles/jaerlang/code/sellaprime_app.erl"erl

-module(sellaprime_app).

-behaviour(application).

-export([start/2, stop/1]).

%%--

%% Function: start(Type, StartArgs) -> {ok, Pid} |

%% {ok, Pid, State} |

%% {error, Reason}

%% Description: This function is called whenever an application

%% is started using application:start/1,2, and should start the processes

%% of the application. If the application is structured according to the

```
%% OTP design principles as a supervision tree, this means starting the

%% top supervisor of the tree.

%%--------------------------------------------------------------------

start(_Type, StartArgs) ->

sellaprime_supervisor:start_link(StartArgs).

%%--------------------------------------------------------------------

%% Function: stop(State) -> void()

%% Description: This function is called whenever an application

%% has stopped. It is intended to be the opposite of Module:start/2 and

%% should do any necessary cleaning up. The return value is ignored.

%%--------------------------------------------------------------------

stop(_State) ->

ok.
```

Здесь должны быть экспортированы функции start/2 и stop/1. Раз мы уже все это сделали, то теперь можем запустить наше приложение в оболочке Эрланга.

```
$ erl -boot start_sasl -config elog3

1> application:loaded_applications().

[{kernel,"ERTS CXC 138 10","2.11.3"},

{stdlib,"ERTS CXC 138 10","1.14.3"},

{sasl,"SASL CXC 138 11","2.1.4"}]

2> application:load(sellaprime).

ok

3> application:loaded_applications().

[{sellaprime,"The Prime Number Shop","1.0"},

{kernel,"ERTS CXC 138 10","2.11.3"},
```

{stdlib,"ERTS CXC 138 10","1.14.3"},

{sasl,"SASL CXC 138 11","2.1.4"}]

4> application:start(sellaprime).

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

ok

5> application:stop(sellaprime).

prime_server stopping

area_server stopping

=INFO REPORT==== 2-Apr-2007::19:34:44 ===

application: sellaprime

exited: stopped

type: temporary

ok

6> application:unload(sellaprime).

ok

7> application:loaded_applications().

[{kernel,"ERTS CXC 138 10","2.11.4"},

{stdlib,"ERTS CXC 138 10","1.14.4"},

{sasl,"SASL CXC 138 11","2.1.5"}]

Вот теперь это вполне оперившееся приложение. Во второй строке мы загрузили приложение; этот вызов загружает весь код, но не запускает приложение. В четвертой строке мы запустили приложение, а в пятой строке остановили его. Заметьте, все видно в распечатке, когда приложения запускаются и останавливаются, соответствующие функции сервера простых чисел и сервера площади были вызваны. В шестой строке мы выгрузили приложение. Все модули приложения были удалены из

памяти.

Когда мы делаем полноценную систему, используя OTP, мы упаковываем её в приложение. Это даёт нам универсальный метод запуска, остановки и управления приложением.

Заметьте, когда мы используем init:stop() для завершения работы системы, то все приложения будут завершены так, как это принято. Это правило хорошего тона.

```
$ erl -boot start_sasl -config elog3

1> application:start(sellaprime).

*** my_alarm_handler init:{xyz,{alarm_handler,[]}}

area_server starting

prime_server starting

ok

2> init:stop().

ok

prime_server stopping

area_server stopping

$
```

Две строки следующие за командой номер 2 получены из сервера площади и сервера простых чисел, они показывают нам, что был вызван метод terminate/2 из модуля обратных вызовов gen_server.

18.8 Организация файловой системы

Я до сих пор ничего не упоминал об организации файловой системы. Это сделано специально - моя цель озадачивать вас проблемами по одной.

Структурированное OTP приложение, обычно, содержит файлы соответствующие различным частям приложения в строго определенных местах. Это не требование; так как все нужные файлы могут быть найдены во время исполнения, но это не потребуется, если все файлы лежат в нужных местах.

Все описанные в этой книге демонстрационные файлы располагаются в одной

директории. Это простейшие примеры, и сделано это во избежание проблем с путями поиска и с взаимодействием между различными програмами.

Основные файлы, используемые в компании sellaprime, следующие:

File

Content

area_server.erl

Сервер областей - модуль обратных вызовов gen_server

prime_server.erl

Сервер простых чисел - модуль обратных вызовов gen_server

sellaprim_supervisor.erl

Модуль обратных вызовов Супервизора

sellaprim_app.erl

Модуль обратных вызовов Приложения

my_alam_handler.erl

Модуль обратных вызовов Событий для gen_event

sellaprime.app

Спецификация приложения

elog4.config

Файл настроек Регистратора ошибок

Для рассмотрения того как используются эти файлы и модули нам нужно рассмотреть последовательность событий, происходящих, когда приложение стартует:

Мы запускаем систему следующими командами: $ erl -boot start_sasl -config elog4.config 1> application:start(sellaprime). ...

Файл sellaprime.app должен находиться в корневом каталоге из которого запускается Эрланг или в подкаталоге этого каталога.

В этом случае контроллер приложения сможет обнаружить {mod, ...}, объявленый в sellaprime.app. Он содержит имя контроллера приложения. И конечно же это модуль

sellaprime_app.

Вызвается обратный вызов sellaprime_app:start/2.

sellaprime_app:start/2 вызывает sellaprime_supervisor:start_link/2, который, в свою очередь, запускает супервизор sellaprime.

Вызвается обратный вызов супервизора sellaprime_supervisor:init/1 - он устанавливает обработчик ошибок и возвращает спецификацию надзора. Спецификация надзора сообщает как запускать сервер площади и сервер простых чисел.

sellaprime супервизор запускает сервер площади и сервер простых чисел. Они реализованы как модули обратных вызовов gen_server.

Останавливать все это очень просто. Вы просто вызываете application:stop(sellaprime) или init:stop().

18.9 Монитор приложений

Монитор приложений - это программа с графическим интерфейсом (GUI) для просмотра запущенных приложений. Команда appmon:start() запускает монитор приложений. Когда вы выполните эту команду, вы увидите окно, похожее на то, которое изображено на рисунке 18.2. Для того чтобы увидеть структуру приложения, вы должны кликнуть по одному из приложений. Монитор приложений для приложения sellaprime показан на рисунке 18.3.

18.10 Копаем глубже

Я пропустил довольно много подробностей, разъяснив только принципы. Вы сможете найти подробности на страницах руководства по gen_event, error_logger, supervisor и application.

18.2 Монитор приложений. Вид при запуске.

Рисунок 18.3 Приложение sellaprime

Следующие файлы содержат более подробную информацию о том как использовать OTP поведения:

HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"httpHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"://HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"wwwHYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK "http://www.erlang.org/doc/pdf/design_principles.pdf"erlangHYPERLINK

"http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"orgHYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"docHYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"pdfHYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"designHYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"_HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"principlesHYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/design_principles.pdf"pdf

(страница 97) Gen servers, gen event, supervisors

HYPERLINK "http://www.erlang.org/doc/pdf/system_principles.pdf"httpHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"://HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"wwwHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"erlangHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"orgHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"docHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"pdfHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"systemHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"_HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"principlesHYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/system_principles.pdf"pdf

(страница 19) Как сделать boot файл

HYPERLINK "http://www.erlang.org/doc/pdf/appmon.pdf"httpHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"://HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"wwwHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"erlangHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"orgHYPERLINK

"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"docHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"pdfHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"/HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"appmonHYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf".HYPERLINK
"http://www.erlang.org/doc/pdf/appmon.pdf"pdf

(страница 16) Монитор приложений

18.11 Как мы вычисляем простые числа?

Очень просто.

%% make a prime with at least K decimal digits.

%% Here we use 'Bertrand's postulate.

%% Bertrands postulate is that for every N > 3,

%% there is a prime P satisfying N < P < 2N - 2

%% This was proved by Tchebychef in 1850

%% (Erdos improved this proof in 1932)

make_prime(1) ->

lists:nth(random:uniform(5), [1,2,3,5,7]);

make_prime(K) when K > 0 ->

new_seed(),

N = make_random_int(K),

if N > 3 ->

io:format("Generating a \~w digit prime " ,[K]),

MaxTries = N - 3,

P1 = make_prime(MaxTries, N+1),

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

```
P1;
true ->
make_prime(K)
end.
make_prime(0, _) ->
exit(impossible);
make_prime(K, P) ->
io:format("." ,[]),
case is_prime(P) of
true -> P;
false -> make_prime(K-1, P+1)
end.
%% Fermat's little theorem says that if
%% N is a prime and if A < N then
%% A\^N mod N = A
is_prime(D) ->
new_seed(),
is_prime(D, 100).
is_prime(D, Ntests) ->
N = length(integer_to_list(D)) -1,
is_prime(Ntests, D, N).
is_prime(0, , ) -> true;
is_prime(Ntest, N, Len) ->
K = random:uniform(Len),
%% A is a random number less than N
```

```
A = make_random_int(K),
if
A < N ->
case lib_lin:pow(A,N,N) of
A -> is_prime(Ntest-1,N,Len);
_ -> false
end;
true ->
is_prime(Ntest, N, Len)
end.
1> lib_primes:make_prime(500).
Generating a 500 digit prime .....
791015726987201027909055597115096126908592921342508297266243912592631402855283461324397013307924771094786030944973946964404399696758714374940531222422946966707622926139385002096578309062534166780603261012226023459181325555764028306928844115181391107802007557066746476035515105154017421267382367314941956505578474497545252666718280976890401503018406521440650857349061213980678938094352667372672691906693169783133618111423622890401868042872198074546193740053777668271056036892838181730070340565057841534Он вернулся!Он вернулся!
```