Глава 16

Введение в OTP

OTP означает Открытая Телекоммуникационная Платформа (Open Telecom Platform). На самом деле это название обманчиво, потому что OTP имеет более широкое применение, чем может показаться. OTP - это часть операционной системы с набором библиотек и процедур, используемых для построения масштабируемых, отказоустойчивых и распределенных приложений. OTP была разработана шведской компанией Ericsson и использовалась внутри Ericsson для разработки отказоустойчивых систем.

OTP содержит ряд мощных инструментов, таких как, полноценный web сервер, FTP сервер, CORBA ORB и других, написанных на Erlang. Еще OTP содержит высокотехнологичные инструменты для создания приложений в сфере телекоммуникаций, с реализацией протоколов H.248, SNMP, и кросс-компилятор ASN.1-to-Erlang. Но я не буду говорить об этом; вы сможете найти информацию по этой теме, посетив сайты, ссылки на которые даны в разделе C.1 Онлайн документации, на странице 399.

Если вы хотите разработать свою программу, используя OTP, тогда основные принципы в поведении OTP будут для вас очень привлекательны. Это поведение объединяет общие поведенческие модели – думайте об этом, как об основе которая, по сути, есть параметризованные вызовы модулей. Мощь OTP исходит из ее свойств, таких как отказоустойчивость, масштабируемость, динамический изменяемый код и т.д. собственно это и есть поведение OTP. Другими словами, при написании обратных вызовов вам не надо беспокоиться об отказоустойчивости, потому что об этом позаботится сама OTP. Java-программисты могут думать о поведении как о J2EE контейнере.

Проще говоря, поведение решает нефункциональную часть проблемы, а обратные вызовы – функциональную. Прелесть в том, что нефункциональная часть проблемы (например, динамическое изменение кода) всегда одинакова для всех приложений, тогда как функциональная часть (реализация обратных вызовов) различна в каждом отдельном случае.

В этой главе мы увидим одно из поведений, модуль gen_server, во всех деталях. Но перед тем как погрузиться во все тонкости работы gen_server, сначала мы рассмотрим простой сервер (простейший сервер, который возможно показать) и будем его изменять шаг за шагом, пока не получим полноценный модуль gen_server. Таким образом, вы реально сможете понять, как работает gen_server и будете готовы к

исследованию внутренностей.

Вот план этой главы:

Написание маленькой клиент-серверной программы на Erlang.

Постепенная «генерализация» этой программы и добавление новых возможностей.

Переход к реальному коду.

16.1 Дорога к типичному (обобщенному) серверу (Generic Server)

Это наиболее важный подраздел в этой книге, прочитайте его один раз, два раза, прочитайте его 100 раз – чтобы убедиться в том, что вы все поняли.

Мы приступаем к написанию четырех маленьких северов с названиями server1**,** server2**....**, каждый слегка будет отличаться от предыдущего. Нашей целью является полное разделение нефункциональной и функциональной частей решаемой задачи. Последнее предложение сейчас скорее всего ничего для вас не значит, но не беспокойтесь – скоро об всем узнаете. Итак, глубоко вдохните...

Сервер №1: Простой сервер

Первая попытка. Это маленький сервер, который мы реализуем, написав модуль обратных вызовов.

```
-module(server1).

-export([start/2, rpc/2]).

start(Name, Mod) ->

register(Name, spawn(fun() -> loop(Name, Mod, Mod:init()) end)).

rpc(Name, Request) ->

Name ! {self(), Request},

receive

{Name, Response} -> Response

end.

loop(Name, Mod, State) ->

receive
```

{From, Request} ->

{Response, State1} = Mod:handle(Request, State),

From ! {Name, Response},

loop(Name, Mod, State1)

end.

Это очень небольшое количество кода является основой сервера. Давайте напишем обратные вызовы для сервера №1. Вот код модуля обратных вызовов:

-module(name_server).

-export([init/0, add/2, whereis/1, handle/2]).

-import(server1, [rpc/2]).

%% client routines

add(Name, Place) -> rpc(name_server, {add, Name, Place}).

whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines

init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};

handle({whereis, Name}, Dict) -> {dict:find(Name, Dict), Dict}.

Этот код фактически выполняет две задачи. Он выступает в роли модуля обратных вызовов, вызываемых из серверного кода, и иногда содержит интерфейсные конструкции, которые будут вызываться на стороне клиента. Обычно, по соглашениям OTP, эти функции объединяются в один модуль.

Чтобы увидеть как это работает, сделайте следующее:

1> server1:start(name_server, name_server).

true

2> name_server:add(joe, "at home").

ok

3> name_server:whereis(joe).

{ok,"at home"}

Сейчас прервемся и подумаем. Обратный вызов не имеет кода для параллелизации, не порождает процессы, не отправляет и не принимает сообщения, ничего не регистрирует. Это просто последовательный код и ничего более. Что же это значит? А это означает то, что мы сможем написать клиент-серверное приложение без понимания того, что лежит в основе модели параллельных процессов. Это основной шаблон для всех серверов. Однажды вы поймете основные структуры, это просто как «цигарка».

Сервер №2: Сервер с транзакциями

В этом примере сервер прервет клиента, если результатом запроса будет ошибка:

```
-module(server2).

-export([start/2, rpc/2]).

start(Name, Mod) ->

register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

rpc(Name, Request) ->

Name ! {self(), Request},

receive

{Name, crash} -> exit(rpc);

{Name, ok, Response} -> Response

end.

loop(Name, Mod, OldState) ->

receive

{From, Request} ->

try Mod:handle(Request, OldState) of

{Response, NewState} ->

From ! {Name, ok, Response},

loop(Name, Mod, NewState)
```

catch

_:Why ->

log_the_error(Name, Request, Why),

%% send a message to cause the client to crash

From ! {Name, crash},

%% loop with the *original* state

loop(Name, Mod, OldState)

end

end.

log_the_error(Name, Request, Why) ->

io:format("Server \~p request \~p \~n"

"caused exception \~p\~n" ,

[Name, Request, Why]).

Единственное что дает нам «транзакционную семантику» в этом сервере – это цикл с *оригинальным значением* State**,** если возникает исключение в обработчике. Но если обработчик завершится успешно, то тогда цикл со значением NewState**,** которое предоставляет обработчик.

Зачем хранить оригинальное состояние? Обработчик выполняется с ошибкой тогда, когда клиент отправляет неверное сообщение, в ответ клиент получает сообщение об аварии. Клиент не может работать, потому что запрос, отправленный на сервер, привел к сбою в обработчике. Зато другие клиенты желающие использовать этот сервер не пострадают. Более того, состояние сервера не изменится, когда в обработчике возникнет ошибка.

Замечу, что модуль обратных вызовов для этого сервера точно такой же как и для сервера №1. Изменяя сервер и оставляя неизменным модуль обратных вызовов, мы может менять нефункциональную часть поведения модуля обратных вызовов.

Примечание: Последнее высказывание не является чистой правдой. Мы все-таки сделали небольшие изменения в модуле обратных вызовов, когда мы перешли от сервера №1 к серверу №2 мы все же изменили имя сервера в директиве –import с server1 на server2. Других изменений не было.

Сервер №3: Сервер с горячей заменой кода

Сейчас мы добавим в наш сервер механизм горячей замены кода:

```
-module(server3).

-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->

register(Name,

spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->

Name ! {self(), Request},

receive

{Name, Response} -> Response

end.

loop(Name, Mod, OldState) ->

receive

{From, {swap_code, NewCallBackMod}} ->

From ! {Name, ack},

loop(Name, NewCallBackMod, OldState);

{From, Request} ->

{Response, NewState} = Mod:handle(Request, OldState),

From ! {Name, Response},

loop(Name, Mod, NewState)

end.
```

Как же это работает?

Если мы отправляем серверу сообщение о замене кода (swap code), значит мы хотим

заменить работающий модуль обратных вызовов на новый модуль, имя которого передается в сообщении. Продемонстрировать это можно запустив server3 с модулем обратных вызовов и динамический подменить модуль на новый. Мы не сможем использовать name_server в качестве модуля обратных вызовов, поскольку это имя сервера и оно жестко задано, так как компилируется внутрь модуля сервера. В итоге нам необходимо сделать копию старого модуля и назвать его name_server1, где мы изменим имя сервера:

```
-module(name_server1).

-export([init/0, add/2, whereis/1, handle/2]).

-import(server3, [rpc/2]).

%% client routines

add(Name, Place) -> rpc(name_server, {add, Name, Place}).

whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines

init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};

handle({whereis, Name}, Dict) -> {dict:find(Name, Dict), Dict}.
```

Сначала мы запустим server3 с модулем обратных вызовов name_server1:

```
1> server3:start(name_server, name_server1).

true

2> name_server:add(joe, "at home").

ok

3> name_server:add(helen, "at work").

ok
```

Теперь, я полагаю, мы захотим найти все имена которые обслуживает наш сервер имен. Но в нашем API нет функции для выполнения такой задачи – модуль name_server имеет лишь функции для добавления и поиска имен.

Молниеносно запускаем наш текстовый редактор и пишем наш новый модуль обратных

вызовов:

```
-module(new_name_server).

-export([init/0, add/2, all_names/0, delete/1, whereis/1, handle/2]).

-import(server3, [rpc/2]).

%% interface

all_names() -> rpc(name_server, allNames).

add(Name, Place) -> rpc(name_server, {add, Name, Place}).

delete(Name) -> rpc(name_server, {delete, Name}).

whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines

init() -> dict:new().

handle({add, Name, Place}, Dict) -> {ok, dict:store(Name, Place, Dict)};

handle(allNames, Dict) -> {dict:fetch_keys(Dict), Dict};

handle({delete, Name}, Dict) -> {ok, dict:erase(Name, Dict)};

handle({whereis, Name}, Dict) -> {dict:find(Name, Dict), Dict}.
```

Сейчас мы скомпилируем этот код и скажем серверу заменить работающий модуль обратных вызовов новым:

```
4> c(new_name_server).

{ok,new_name_server}

5> server3:swap_code(name_server, new_name_server).

Ack
```

И можем запустить новые функции это сервера:

```
6> new_name_server:all_names().

[joe,helen]
```

Здесь мы *заменили модуль обратных вызовов «на лету»* - это и есть динамическая замена кода в действии, вы все видели сами и никакой черной магии.

Сейчас прервемся и снова подумаем. Последние две задачи, которые мы с вами решили, в целом, считаются сложными, но на самом деле это очень сложные задачи. Серверы с механизмом транзакций сложны в написании; серверы с динамической заменой кода еще более сложны в написании.

Эта технология чрезвычайно мощна. Обычно мы думаем о серверах как о программах, чье состояние меняется при отправке им сообщений. Код в серверах фиксированный при первом их запуске, и если мы хотим изменить поведение сервера, то нам необходимо остановить сервер, изменить его код и снова его запустить. В этом примере, код сервера может быть изменен также легко, как можно изменить состояние у сервера.

Сервер №4: Транзакции и горячая замена кода

В предыдущих двух серверах семантика горячей замены и семантика транзакций были разделены. Давайте объединим обе возможности в одном сервере. Итак, держите ваши шляпы…

```
-module(server4).

-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->

register(Name, spawn(fun() -> loop(Name,Mod,Mod:init()) end)).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->

Name ! {self(), Request},

receive

{Name, crash} -> exit(rpc);

{Name, ok, Response} -> Response

end.

loop(Name, Mod, OldState) ->

receive

{From, {swap_code, NewCallbackMod}} ->

From ! {Name, ok, ack},
```

```
loop(Name, NewCallbackMod, OldState);

{From, Request} ->

try Mod:handle(Request, OldState) of

{Response, NewState} ->

From ! {Name, ok, Response},

loop(Name, Mod, NewState)

catch

_: Why ->

log_the_error(Name, Request, Why),

From ! {Name, crash},

loop(Name, Mod, OldState)

end

end.

log_the_error(Name, Request, Why) ->

io:format("Server \~p request \~p \~n"

"caused exception \~p\~n" ,

[Name, Request, Why]).
```

Этот сервер предоставляет обе возможности, и горячую замену кода и транзакции. Замечательно.

Сервер №5: Еще больше кайфа

Теперь, получив знания о динамической замене кода, мы можем кайфануть по полной программе. Сейчас мы рассмотрим сервер, который ничего не делает, пока мы ему не скажем, изменить поведение:

```
-module(server5).

-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).
```

```
wait() ->
receive
{become, F} -> F()
end.
rpc(Pid, Q) ->
Pid ! {self(), Q},
receive
{Pid, Reply} -> Reply
end.
```

Если мы запустим это сервер и отправим ему сообщение {become, F}, то он превратится в F сервер, исполнив F(). Запустим сервер:

```
1> Pid = server5:start().
<0.57.0>
```

Наш сервер ничего не делает, он просто ждет сообщение become.

Теперь давайте создадим функциональность сервера. Ничего сложного придумывать не будем, просто посчитаем факториал:

```
-module(my_fac_server).
-export([loop/0]).
loop() ->
receive
{From, {fac, N}} ->
From ! {self(), fac(N)},
loop();
{become, Something} ->
Something()
end.
```

```
fac(0) -> 1;

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

Эрланг в PlanetLab

Несколько лет назад, когда мои исследования только начинались, я работал в PlanetLab. Я имел доступ к сети PlanetLab и установил «пустые» Эрланг серверы на все компьютеры (около 450-ти машин). Я не знал что я буду делать с этими машинами, просто установил серверную инфраструктуру для использования в каких-нибудь целях в будущем.

Так как я запустил серверы, я мог легко сказать, пустым серверам превратиться в серверы, выполняющие реальную работу.

Обычная практика (для начала) – это запустить web-серверы, и установить плагины на web-серверы. Мой подход – отступить на один шаг назад и установить пустые серверы. Потом уже устанавливать плагины web-серверов для превращения пустых серверов в web-серверы. Ведь когда web-сервер станет не нужен, мы можем заставить серверы выполнять что-нибудь еще.

Скомпилируйте этот код, теперь вы сможете сказать процессу <0.57.0>**,** превратиться в факториал-сервер:

```
2> c(my_fac_server).

{ok,my_fac_server}

3> Pid ! {become, fun my_fac_server:loop/0}.

{become,#Fun<my_fac_server.loop.0>}
```

Теперь, когда наш сервер стал факториал-сервером, мы сделаем вызов:

```
4> server5:rpc(Pid, {fac,30}).

265252859812191058636308480000000
```

Наш процесс будет факториал-сервером до тех пор, пока мы не скажем ему стать кем-нибудь другим, отправив ему сообщение {become, Something}.

Как вы увидели в предыдущем примере, мы может иметь широкий диапазон различных типов серверов, с различной семантикой и совершенно удивительными свойствами. Эта технология почти такая же мощная. Используя весь ее потенциал, можно создавать очень маленькие программы удивительной мощности и красоте. Когда мы

делали проекты промышленных масштабов с десятками и сотнями программистов, мы не хотели делать некоторые вещи слишком динамичными. Мы хотели добиться баланса между обобщенностью и мощностью и получали нечто подходящее для коммерческих продуктов. Получали код, который меняется от версии к версии, и который превосходно работает, но очень сложен в отладке, если вдруг что-то пойдет не так. Если мы делали много динамических изменений в нашем коде, и это переставало работать, найти причину было очень нелегко.

Примеры серверов в этом разделе не совсем корректны. Они писались, чтобы показать идеи, но они содержат одну или две чрезвычайно маленьких и тонких ошибки. Я не буде прямо сейчас рассказывать о них, я дам некоторые комментарии по этому поводу в конце главы.

Эрланговый модуль gen_server – это что-то вроде логического завершения последовательности достаточно простых серверов (точно таких же которые мы писали на протяжении всей главы).

Он используется в промышленных продуктах, начиная с 1998 года. Сотни серверов могут быть частью одного продукта. Эти серверы будут написаны программистами с использованием обычного последовательного кода. Все ошибки обрабатываются, и все нефункциональное поведение учтено в типовой части сервера.

Итак, сейчас мы совершим огромный прыжок и рассмотрим gen_server.

16.2 Начнем с gen_server

Я собираюсь окунуть вас в самую глубь проблемы. Вот простой план написания модуля обратных вызовов для gen_server, состоящий из трех пунктов:

\1. Выбрать имя для модуля обратных вызовов.

\2. Написать интерфейсные функции.

\3. Написать шесть обязательных функций для модуля обратных вызовов.

На самом деле это очень просто. Не думайте – просто следуйте плану!

Шаг 1: Выбрать имя для модуля обратных вызовов

Мы будем делать очень простую платежную систему. Поэтому назовем модуль my_bank.

Шаг 2: Написать интерфейсные конструкции

Мы определим пять интерфейсных конструкций, все они будут в модуле my_bank:

start()

Открыть банк.

stop()

Закрыть банк.

new_account(Who)

Создать новый аккаунт.

deposit(Who, Amount)

Положить деньги в банк.

withdraw(Who, Amount)

Взять деньги, если есть на счету.

Каждая конструкция это ровно одна конструкция для вызова gen_server, как показано ниже:

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

stop() -> gen_server:call(?MODULE, stop).

new_account(Who) -> gen_server:call(?MODULE, {new, Who}).

deposit(Who, Amount) -> gen_server:call(?MODULE, {add, Who, Amount}).

withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).

gen_server:start_link({local, Name}, Mod, ...) запускает *локальный сервер* (можно использовать аргумент global для того, чтобы он был доступен кластеру Эрланг-серверов). Макрос ?MODULE содержит имя модуля my_bank. Mod – это имя модуля обратных вызовов. Остальные аргументы gen_server:start мы пока не будем рассматривать.

gen_server:call(?MODULE, Term) используется для вызова удаленных процедур сервера.

Шаг 3: Написать конструкции модуля обратных вызовов

Наш модуль обратных вызовов должен экспортировать шесть функций: init/1,

handle_call/3, handle_cast/2, handle_info/2, terminate/2, и code_change/3.

Чтобы облегчить жизнь, мы можем использовать один из шаблонов для создания gen_server. Вот пример:

```
%% gen_server_mini_template

-behaviour(gen_server).

-export([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, ?SERVER}, ?MODULE, [], []).

init([]) -> {ok, State}.

handle_call(Request, From, State) -> {reply, Reply, State}.

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

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

terminate(Reason, State) -> ok.

code_change(_OldVsn, State, Extra) -> {ok, State}.
```

Этот пример содержит простой скелет, который нужно заполнить, чтобы получить сервер. Ключевое слово –behaviour используется компилятором, чтобы знать какие предупреждения и сообщения об ошибках генерировать.

Примечание: Если вы используете emacs, то вы сможете вставить шаблон несколькими командами. Если ваш редактор переключен в режим эрланга, то выберите в меню Erlang -> Skeletons для создания шаблона gen_server. Если у вас нет emacs, то не паникуйте. Я включил текст шаблона в конец главы.

Итак, шаблон вставлен, и мы просто отредактируем его куски. Мы имеем все аргументы в интерфейсных конструкциях, согласно аргументам шаблона.

Наиболее важная часть для нас это функция handle_call/3. Мы реализуем код трех запросов в нашем интерфейсе. Пока заполним многоточиями некоторые места, как показано ниже:

```
handle_call({new, Who}, From, State} ->
```

Reply = ...

State1 = ...

{reply, Reply, State1};

handle_call({add, Who, Amount}, From, State} ->

Reply = ...

State1 = ...

{reply, Reply, State1};

handle_call({remove, Who, Amount}, From, State} ->

Reply = ...

State1 = ...

{reply, Reply, State1};

Значение Reply отправляется обратно клиенту, как результат вызова удаленной процедуры.

State это просто переменная, представляющая глобальное состояние сервера, оно было передано серверу. В нашем банковском сервере состояние не меняется; это просто индекс ETS таблицы и он постоянный (хотя содержимое таблицы меняется).

После редактирования кусков кода в шаблоне, мы получили следующий код:

init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->

Reply = **case** ets:lookup(Tab, Who) **of**

[] -> ets:insert(Tab, {Who,0}),

{welcome, Who};

[_] -> {Who, you_already_are_a_customer}

end,

{reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->

```
Reply = case ets:lookup(Tab, Who) of
[] -> not_a_customer;
[{Who,Balance}] ->
NewBalance = Balance + X,
ets:insert(Tab, {Who, NewBalance}),
{thanks, Who, your_balance_is, NewBalance
end,
{reply, Reply, Tab};
handle_call({remove,Who, X}, _From, Tab) ->
Reply = case ets:lookup(Tab, Who) of
```

```
[] -> not_a_customer;
[{Who,Balance}] when X =< Balance ->
NewBalance = Balance - X,
ets:insert(Tab, {Who, NewBalance}),
{thanks, Who, your_balance_is, NewBalance};
[{Who,Balance}] ->
{sorry,Who,you_only_have,Balance,in_the_bank}
end,
{reply, Reply, Tab};
handle_call(stop, _From, Tab) ->
{stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(Reason, State) -> ok.
```

code_change(_OldVsn, State, Extra) -> {ok, State}.

Запускаем наш сервер, вызвав gen_server:start_link(Name, CallBackMod, StartArgs, Opts); эта конструкция вызовет в модуле обратных вызовов Mod:init(StartArgs), и должны нам вернуть {ok, State}. Значение State передается как третий аргумент в handle_call.

Отмечу как мы остановим сервер. handle_call(Stop, From, Tab) вернет {stop, normal, stopped, Tab} при остановке сервера. Второй аргумент (normal) используется как первый аргумент в конструкции my_bank:terminate/2. Третий аргумент (stopped) становится возвращаемым значением my_bank:stop().

Теперь все готово. Давайте посетим наш банк:

1> my_bank:start().

{ok,<0.33.0>}

2> my_bank:deposit("joe", 10).

not_a_customer

3> my_bank:new_account("joe").

{welcome,"joe"}

4> my_bank:deposit("joe", 10).

{thanks,"joe",your_balance_is,10}

5> my_bank:deposit("joe", 30).

{thanks,"joe",your_balance_is,40}

6> my_bank:withdraw("joe", 15).

{thanks,"joe",your_balance_is,25}

7> my_bank:withdraw("joe", 45).

{sorry,"joe",you_only_have,25,in_the_bank}

16.3 Структура обратных вызовов gen_server

Теперь вооружившись идеями, можем приступить к более детальному рассмотрению структуры обратных вызовов gen_server.

Что же происходит, когда мы запускаем сервер?

Вызов gen_server:start_link(Name, Mod, InitArgs, Opts) запускает все. Создается сервер Name. Запускается модуль обратных вызовов Mod. Opts управляют поведением типичного сервера. Здесь может быть протоколирование сообщений, функции отладки, и много чего еще. Типичный сервер запускается вызовом Mod:init(InitArgs).

Ниже приведен шаблон для init:

%%--

%% Function: init(Args) -> {ok, State} |

%% {ok, State, Timeout} |

%% ignore |

%% {stop, Reason}

%% Description: Initiates the server

%%--

init([]) ->

{ok, #state{}}.

При нормальном положении дел, мы просто вернем {ok, State}. Значение других аргументов вы можете найти в руководстве по gen_server.

Если возвращается значение {ok, State}, значит, сервер успешно запущен и его начальное состояние State.

Что же происходит, когда мы обращаемся к серверу?

Для обращения к серверу клиентская программа вызывает gen_server:call(Name, Request). В результате будет вызвана функция handle_call/3 из модуля обратных вызовов.

handle_call/3 имеет следующий шаблон:

%%--

%% Function:

%% handle_call(Request, From, State) -> {reply, Reply, State} |

%%   {reply, Reply, State, Timeout} |

%% {noreply, State} |

%% {noreply, State, Timeout} |

%% {stop, Reason, Reply, State} |

%% {stop, Reason, State}

%% Description: Handling call messages

%%--

handle_call(*Request,* From, State) ->

Reply = ok,

{reply, Reply, State}.

Request (второй аргумент gen_server:call/2) станет первым аргумент handle_call/3. From – это PID, запрашивающего клиентского процесса, а State – это текущее состояние клиента.

Если все хорошо, мы возвращаем {reply, Reply, NewState}. Когда это происходит Reply уходит обратно к клиенту, где превращается в возвращаемое значение gen_server:call. NewState – это следующее состояние сервера.

Другие возвращаемые значения {noreply, …} и {stop, …} используются достаточно редко, noreply заставляет сервер продолжить работу, но клиент будет ожидать ответа, так как сервер озадачен отвечать всем клиентам. Вызов stop с соответствующими аргументами остановит сервер.

Вызовы и Образы

Мы увидели взаимодействие между gen_server:call и handle_call. Это то, что используется для реализации *вызова удаленных процедур (remote procedure call)*. gen_server:cast(Name, Name) реализация образа, который просто вызывается, не возвращая значений (на самом деле просто сообщение, но обычно это вызов образа из удаленной процедуры).

Соответствующий обратный вызов handle_cast показан в шаблоне ниже:

%%--

%% Function: handle_cast(Msg, State) -> {noreply, NewState} |

%% {noreply, NewState, Timeout} |

%% {stop, Reason, NewState}

%% Description: Handling cast messages

%%--

handle_cast(_Msg, State) ->

{noreply, NewState}.

Обработчик обычно возвращает {noreply, NewState}, который меняет состояние сервера или {stop, …}, который останавливает сервер.

Спонтанные сообщения

Функция обратного вызова handle_info(info, State) используется для обработки спонтанных сообщений получаемых сервером. Так что же такое спонтанные сообщения? Если сервер связан с другими процессами и перехватывает их, он может внезапно принять неожиданное сообщение {‘EXIT’, Pid, What}. Либо, любой процесс в системе, который знает о PID сервера, может просто отправить ему сообщение. Любые такие сообщения будут приняты сервером как значение переменной Info.

Шаблон для handle_info выглядит так:

%%--

%% Function: handle_info(Info, State) -> {noreply, State} |

%% {noreply, State, Timeout} |

%% {stop, Reason, State}

%% Description: Handling all non-call/cast messages

%%--

handle_info(_Info, State) ->

{noreply, State}.

Возвращаемое значение такое же как у handle_cast.

Прощай, малышка.

Сервер может прервать свою работу по многим причинам. Один из handle_Something вызовов может вернуть {stop, Reason, NewState}, либо сервер может рухнуть при сообщении {‘Exit’, reason}. При любом раскладе будет вызвана функция terminate(Reason, NewState).

Вот ее шаблон:

%%--

%% Function: terminate(Reason, State) -> void()

%% Description: This function is called by a gen_server when it is

%% about to terminate. It should be the opposite of Module:init/1 and

%% do any necessary

%% cleaning up. When it returns, the gen_server terminates with Reason.

%% The return value is ignored.

%%--

terminate(_Reason, State) ->

ok.

Эта функция не возвращает новое состояние, потому что вся работа уже прервана. И что же можно сделать в этим состоянием (State)? Оказывается многое. Мы можем сохранять его на диск, Отправить в сообщении другим процессам или отказаться от него, если это необходимо приложению. Если вы хотите чтобы ваш сервер был когда-нибудь перезапущен, вам необходимо написать функцию «Я еще вернусь», которую вызовет terminate/2.

Замена кода.

Вы можете динамически изменять состояние своего сервера, пока он запущен. Вызов этой функции обратного вызова производит подсистема "управления релизами" когда система выполняет обновление программного кода.

Эта часть детально описана в разделе "Управление релизами" в документации о принципах дизайна OTP.

16.4 Код и Шаблоны

Этот код сделан в emacs:

gen_server template

%%%--

%%% File : gen_server_template.full

```
%%% Author : my name yourname@localhost.localdomain
%%% Description :
%%%
%%% Created : 2 Mar 2007 by my name yourname@localhost.localdomain
%%%-------------------------------------------------------------------
-module().
-behaviour(gen_server).
%% API
-export([start_link/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
terminate/2, code_change/3]).
-record(state, {}).
%%====================================================================
%% API
%%====================================================================
%%--------------------------------------------------------------------
%% Function: start_link() -> {ok,Pid} | ignore | {error,Error}
%% Description: Starts the server
%%--------------------------------------------------------------------
start_link() ->
gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
%%====================================================================
%% gen_server callbacks
%%====================================================================
```

```
%%--------------------------------------------------------------------
%% Function: init(Args) -> {ok, State} |
%% {ok, State, Timeout} |
%% ignore |
%% {stop, Reason}
%% Description: Initiates the server
%%--------------------------------------------------------------------
init([]) ->
{ok, #state{}}.
%%--------------------------------------------------------------------
%% Function: %% handle_call(Request, From, State) -> {reply, Reply, State} |
%% {reply, Reply, State, Timeout} |
%% {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, Reply, State} |
%% {stop, Reason, State}
%% Description: Handling call messages
%%--------------------------------------------------------------------
handle_call(Request, From, State) ->
Reply = ok,
{reply, Reply, State}.
%%--------------------------------------------------------------------
%% Function: handle_cast(Msg, State) -> {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, State}
```

```
%% Description: Handling cast messages
%%--------------------------------------------------------------------
handle_cast(_Msg, State) ->
{noreply, State}.
%%--------------------------------------------------------------------
%% Function: handle_info(Info, State) -> {noreply, State} |
%% {noreply, State, Timeout} |
%% {stop, Reason, State}
%% Description: Handling all non call/cast messages
%%--------------------------------------------------------------------
handle_info(_Info, State) ->
{noreply, State}.
%%--------------------------------------------------------------------
%% Function: terminate(Reason, State) -> void()
%% Description: This function is called by a gen_server when it is about to
%% terminate. It should be the opposite of Module:init/1 and do any necessary
%% cleaning up. When it returns, the gen_server terminates with Reason.
%% The return value is ignored.
%%--------------------------------------------------------------------
terminate(Reason, State) ->
ok.
%%--------------------------------------------------------------------
%% Func: code_change(OldVsn, State, Extra) -> {ok, NewState}
%% Description: Convert process state when code is changed
%%--------------------------------------------------------------------
```

code_change(*OldVsn, State,* Extra) ->

{ok, State}.

%%--

%%% Internal functions

%%--

my_bank

-**module**(my_bank).

-**behaviour**(gen_server).

-**export**([start/0]).

%% gen_server callbacks

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

terminate/2, code_change/3]).

-**compile**(export_all).

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

stop() -> gen_server:call(?MODULE, stop).

new_account(Who) -> gen_server:call(?MODULE, {new, Who}).

deposit(Who, Amount) -> gen_server:call(?MODULE, {add, Who, Amount}).

withdraw(Who, Amount) -> gen_server:call(?MODULE, {remove, Who, Amount}).

init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->

Reply = **case** ets:lookup(Tab, Who) of

[] -> ets:insert(Tab, {Who,0}),

{welcome, Who};

[_] -> {Who, you_already_are_a_customer}

```
end,
{reply, Reply, Tab};
handle_call({add,Who,X}, _From, Tab) ->
Reply = case ets:lookup(Tab, Who) of
[] -> not_a_customer;
[{Who,Balance}] ->
NewBalance = Balance + X,
ets:insert(Tab, {Who, NewBalance}),
{thanks, Who, your_balance_is, NewBalance}
end,
{reply, Reply, Tab};
handle_call({remove,Who, X}, _From, Tab) ->
Reply = case ets:lookup(Tab, Who) of
[] -> not_a_customer;
[{Who,Balance}] when X =< Balance ->
NewBalance = Balance - X,
ets:insert(Tab, {Who, NewBalance}),
{thanks, Who, your_balance_is, NewBalance};
[{Who,Balance}] ->
{sorry,Who,you_only_have,Balance,in_the_bank}
end,
{reply, Reply, Tab};
handle_call(stop, _From, Tab) ->
{stop, normal, stopped, Tab}.
handle_cast(_Msg, State) -> {noreply, State}.
```

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

terminate(*Reason,* State) -> ok.

code_change(_OldVsn, State, Extra) -> {ok, State}.

16.5 Копаем глубже

Мы увидели, что gen_server достаточно прост. Мы не рассмотрели некоторые интерфейсные функции gen_server-а, и не поговорили обо всех аргументах интерфейсных функций. Если вы поняли основные идеи, то с деталями разберетесь, обратившись к документации по gen_server.

В этой главе мы увидели только простейшие возможные пути использования gen_server, но этого должно быть достаточно для решения большинства задач. Более сложные приложения позволяют gen_server-у отвечать со значением noreply и и делегировать ответ другому процессу. Информацию об этом вы можете прочитать в главе "Принципы дизайна" и в руководстве по модулям sys и proc_lib.

Ericsson has released OTP subject to the Erlang Public License (EPL). EPL is a derivative of the Mozilla Public License (MPL).

**

Я использовал эту технологию во многих продуктах, которые никогда не останавливались для модернизации кода.

Availiable from http://www.erlang.org/doc/pdf/design_principles.pdf.

http://media.pragprog.com/titles/jaerlang/code/gen_server_template.full

http://media.pragprog.com/titles/jaerlang/code/my_bank.erl

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