Фреймворк OTP

## Общий подход

OTP — фреймворк, предоставляющий обобщённые реализации типичных паттернов Erlang, называемых поведениями.

OTP делит процессы на супервизор и рабочие. Поведения супервизора:

1. supervisor.

Поведения для рабочих процессов:

1. gen\_event — реализация обработчика событий;
2. gen\_server — реализация сервера для паттерна «клиент-сервер»;
3. gen\_statem — конечный автомат, получающий данные через сообщения.

Принцип работы с поведениями схож: в модуле поведения определены общие алгоритмы для соответствующего паттерна. Задача программиста состоит в том, чтобы написать модуль, содержащий callback-функции для какого-то определённого случая использования — обработчики состояния, обработчики методов API сервера и т.д.

Дальнейшее повествование будет строится по следующему принципу:

* Общее описание взаимодействия разработчика с указанным поведением OTP;
* Детальное описание параметров функций, предоставляемых в модуле поведения.

## Application

*«When you have written code implementing some specific functionality you might want to make the code into an application, that is, a component that can be started and stopped as a unit, and which can also be reused in other systems.»*

Приложение — самостоятельная часть кода, которую возможно переиспользовать и которая структурно является OTP-деревом (вершина — супервизор). Каждое приложение имеет конфигурацию — **resource-файл,**  который содержит все необходимые переменные окружения.

Приложения могут располагаться внутри релиза, проект в котором находятся коды релиза называют зонтичным.

**ЗАПУСК**

Существует несколько методов, обеспечивающих запуск приложения:

|  |  |  |
| --- | --- | --- |
| **Функция + аргументы** | **Выходное значение** | **Примечание** |
| start(Application, [Type]) | The function initialises everything for your app and only needs to return the Pid of the application's top-level supervisor in one of the two following forms:  {ok, Pid} or  {ok, Pid, SomeState}.  If you don't return SomeState, it simply defaults to []. | Запускает приложение, если оно не запущено.  Перед стартом проверяет наличие запущенных зависимостей, если кого-то не хватает, то исключение.  Type:   * **temporary** (по умолчанию) — при завершении посылается отчёт, остальные приложения продолжают работу * **permanent —** при завершении забирает за собой все приложения текущей ноды erlang; * **transient —** если причина завершения *normal,* то остальные приложения продолжают работу, иначе — приложение забирает за собой все приложения текущей ноды. |
| ensure\_all\_started(Application) | **{ok, Started} | {error, Reason}** | Рекурсивный старт приложения |
| ensure\_started(Application, Type) | ok | {error, Reason} | Equivalent to [start/1,2](https://www.erlang.org/doc/man/application.html" \l "start-1) except it returns ok for already started applications. |
|  |  |  |

**ОСТАНОВКА**

Функция **stop.** Она завершает работу одного приложения вручную, не затрагивая остальных.

По сути своей релиз — приложение приложений.

## supervisor

Супервизор ответственен за запуск, остановку и мониторинг процессов-потомков. Центральная идея супервизора состоит в том, что он должен держать потомков живыми, перезапуская их по необходимости.

Принцип работы с supervisor:

1. Программист в модуле кастомного супервизора описывает функцию старта супервизора, которая под капотом вызывает supervisor:start\_link/2;
2. Программист в модуле кастомного супервизора описывает функцию init/1. Функция должна вернуть две спецификации:
   1. Спецификация супервизора;
   2. Спецификация потомков.

### Старт

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| start\_link(SupName, Module, Args) | {ok, pid()} | ignore | {error, startlink\_err()} | Стартует процесс-наблюдатель, указывая модуль в котором находятся все требуемые callback-функции |

Входные аргументы полностью аналогичны функции gen\_server:start (см. Инициализация).

### Инициализация

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| init(Args) | {ok, {SupervisorSpecification, ChildSpecification}} | Формирует спецификацию будущего супервизора на основе полученных аргументов. Спецификация определяет поведение супервизора при падении потомков. |

#### Спецификация супервизора

Спецификация супервизора определяет общую стратегию при падении одного из потомков. Спецификация описывается кортежем:

{RestartStrategy, AllowedRestarts, MaxSeconds},

Где:

* RestartStrategy — определяет как завершение одного из процессов повлияет на остальные:
  + one\_for\_one — перезапуск только упавшего процесса. Процессы слабо зависят друг от друга;
  + one\_for\_all — завершение и перезапуск всех процессов при падении одного потомка. Все процессы сильно связаны друг с другом;
  + rest\_for\_all — завершение и перезапуск всех потомков, которые были запущены после упавшего процесса. Процессы друг от друга сильно зависят слева-направо в порядке запуска.
* AllowedRestarts — максимальное число перезапусков;
* MaxSeconds — промежуток времени в течение которого работает ограничение AllowedRestarts.

#### Спецификация потомков

Спецификация потомков — указывает супервизору какие процессы должны быть запущены, в каком порядке, и как наблюдатель должен перезапускать и завершать каждый из них.

Спецификация потомков описывается массивом кортежей вида:

{Id, MFA, Restart, Shutdown, Type, ModuleList},

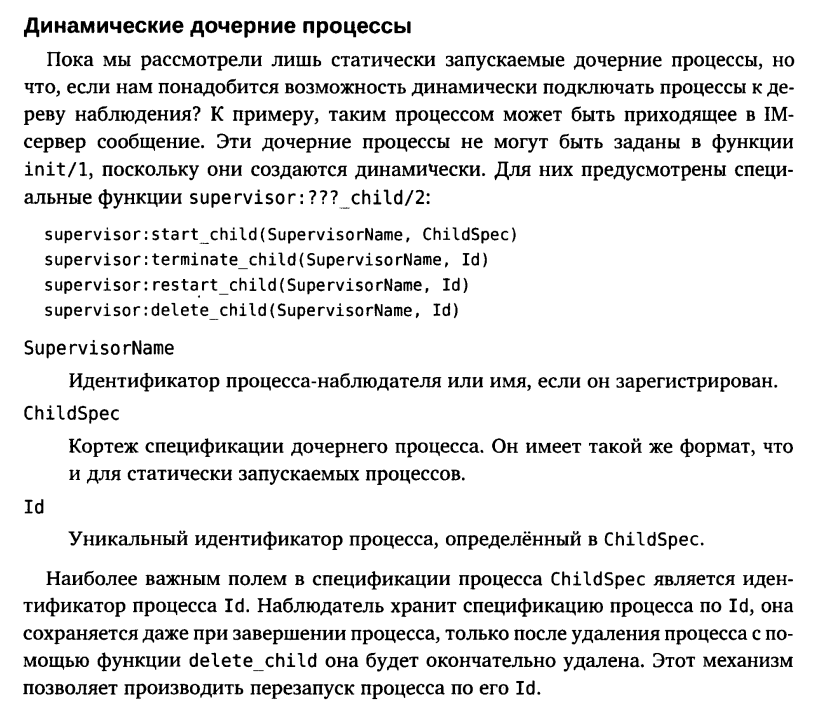
Где:

* Id — уникальный идентификатор процесса среди дочерних процессов данного наблюдателя. Является обязательным. Под этим идентификатором будет выполняться регистрация дочерних процессов.  
  Таким образом, при перезапуске дочернего процесса нет необходимости модифицировать весь список детей - достаточно просто выполнить регистрацию нового процесса под тем же псевдонимом.
* MFA — кортеж, требуемый для функции spawn/3 — {Module,Function, Arguments};
* Restart — указывает, когда упавший процесс должен быть перезапущен:
  + permanent — процесс перезапускается всегда;
  + temporary — никогда не перезапускается;
  + transient — перезапуск произойдёт только при ненормальном завершении.
* Shutdown — сколько миллисекунд даётся процессу на то, чтобы выполнить свою функцию terminate после получения shutdown-сигнала:
  + terminate — функция освобождения ресурсов процесса, напимер Завершение работы сервера;
  + shutdown-сигнал — как правило, для завершения процесса супервизор посылает потомку сигнал exit с причиной shutdown: exit(Child,shutdown).
* Type — определяет тип процесса:
  + worker;
  + supervisor.
* ModuleList — список модулей в которых определён процесс. Список нужен release-handler (менеджер обновлений), который при обновлении приложений будет знать от каких модулей какие процессы зависят.

В новых версиях фреймворка вместо кортежей допускается использование map-коллекций:

|  |
| --- |
| child\_spec() =  #{  id => child\_id(), % mandatory  start => mfargs(), % mandatory  restart => restart(), % optional  significant => significant(), % optional  shutdown => shutdown(), % optional  type => worker(), % optional  modules => modules()} % optional |

Для проверки спецификации есть функция check\_childspec/1.



## gen\_event

## gen\_server

Принцип работы с gen\_server:

1. Программист в модуле сервера описывает функцию старта сервера, которая под капотом обращается к gen\_server:start\_link.  
   После код в gen\_server будет знать модуль, функции которого позволят заниматься обработкой запросов;
2. Программист в модуле сервера описывает функцию инициализации init/1, которая формирует состояние процесса-сервера;
3. Программист в модуле сервера описывает API, каждая функция которого является обёрткой над асинхронной или синхронной функцией запроса из модуля gen\_server. А каждая посылка запроса к серверу из модуля — функция-обёртка над отправкой сообщения определённого в поведении формата;
4. Программист в модуле сервера описывает функции обработки запросов API — уравнения функций handle\_call или handle\_cast. Если всё сделано верно, то после сопоставления с образцом сервер найдёт нужный обработчик и сможет отправить клиенту ответ.

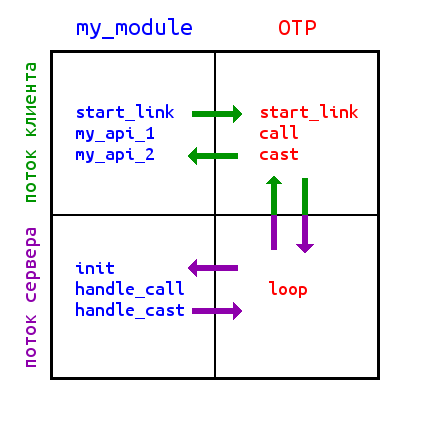


Рисунок 1 — матрица-схема работы с поведением gen\_server

Теперь более детально об API gen\_server.

### Инициализация

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| gen\_server:start(  ServerName::server\_name(),  Module::module(),  Args::term(),  Options::[start\_opt()]) | {ok, Pid} | {error, Reason} | Функция создаёт новый процесс, который вызывает функцию init/1, определённую в модуде Module.  Также есть функция start\_link/4, выполняющая те же действия, но при этом связывающая процесс сервера с тем, откуда был вызван старт. |

Где:

* ServerName::server\_name() — кортеж вида:
  + {local, Name} — сервер зарегистрирует свой процесс локально, используя register/2;
  + {global, Name} — сервер зарегистрирует свой процесс глобально, используя global:register\_name/2:
    - global:register\_name/2 — ассоциация процесса с псевдонима с уведомлением остальных узлов сети, т.е. выполняется межузловая регистрация, когда как register/2 работает в рамках только одного узла.
* Module::module() — атом-имя модуля. В данном случае обозначает модуль, содержащий callback-функции.
* Args::term() — переменная, содержащая входные параметры для функции init/1, определённой в Module
* Options::[start\_opt()]) — флаги, управления памятью, в большинстве случаев достаточно подать функции пустой список.

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| init(Arg) | {ok,LoopData} | Функция, которую разработчик пишет самостоятельно. Нужна для инициализации состояния сервера (аргумента цикла). |

### Синхронное взаимодействие

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| gen\_server:call(Name,Message) | Reply | Синхронная Передача сообщений |

Где:

* Name — имя сервера, определённое при старте, или PID;
* Message — любой терм Erlang, который определён в API сервера.

При получении синхронного сообщения сервер вызовет функцию handle\_call(Message, LoopData), определённую в модуле сервера.

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| handle\_call(Message,From,LoopData) | {reply, Reply, NewLoopData} | {stop, Reason, Reply, NewLoopData} | Обработчик синхронного сообщения. После выполнения отправляет процессу From ответ Reply.  Скорее всего адрес From получается под капотом при вызове gen\_server:call/2  Полученное новое состояние сервера подаётся на вход цикла.  Если функция возврашает stop-кортеж, то сервер вызывает функцию terminate(reason, LoopData) |

### Асинхронное взаимодействие

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| gen\_server:cast(Name,Message) | term() | Асинхронная передача сообщений |

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| handle\_cast(Message,LoopData) | {noreply, NewLoopData} | {stop, Reason, NewLoopData} | Обработчик асинхронного сообщения. Полученное новое состояние сервера подаётся на вход цикла.  Если функция возврашает stop-кортеж, то сервер вызывает функцию terminate(reason, LoopData) |

### Сообщения, не совместимые с OTP

Если сервер получил сообщение неизвестного формата, то тогда он обрабатывает его посредством callback-функции handle\_info(Message, LoopData).

### Завершение работы сервера

При завершении своей работы сервер вызывает callback-функцию terminate/2, которую программист должен определить в модуле сервера самостоятельно. По сути, она является обратной по отношению к init/1 и нужна для освобождения выделенных ресурсов.

Функция является опциональной.

| **Функция** | **Выход** | **Описание** |
| --- | --- | --- |
| terminate(Reason, LoopData) |  | Освобождает выделенные сервером ресурсы. После сервер завершает выполнение своего процесса. |

Где:

* Reason = normal | shutdown | {shutdown, term()} | term()

If the gen\_server process is part of a supervision tree and is ordered by its supervisor to terminate, this function is called with Reason=shutdown if the following conditions apply:

The gen\_server process has been set to trap exit signals.

The shutdown strategy as defined in the child specification of the supervisor is an integer time-out value, not brutal\_kill.

Even if the gen\_server process is not part of a supervision tree, this function is called if it receives an 'EXIT' message from its parent. Reason is the same as in the 'EXIT' message.

## gen\_state