Erlang/OTP

VOLUMEN II
Las bases de OTP

MANUEL ÁNGEL RUBIO JIMÉNEZ

Erlang/OTP

Volumen II: Las bases de OTP

MANUEL ÁNGEL RUBIO JIMÉNEZ

El desarrollo en Erlang se fundamenta en dos bases bien definidas por su autor Joe Armstrong. La primera son la potencia de los procesos que implementa la máquina virtual de Erlang y la segunda es la metodología de Programación Orientada a la Concurrencia que se facilita con el framework OTP.

El uso de Erlang queda incompleto si no se emplean ambas cosas. Su potencia no se descubre hasta no hacer uso de los procesos como fuente principal de programación, y de los *comportamientos* que integra el framework OTP. Para la realización de un proyecto profesional es indispensable el conocimiento y dominio de esta tecnología.

Este segundo volumen de Erlang/OTP cubre el conocimiento de este framework, la creación de proyectos profesionales con él y la Programación Orientada a la Concurrencia o Modelo Actor como se conoce más extensamente. Termina el recorrido necesario para conocer las potencias del lenguaje y de su plataforma. Da al lector un recorrido por la teoría y la práctica acercando aún más herramientas y más código útil para lanzarse a desarrollar.

Revisión Técnica

José Luis Gordo Romero | Juan Sebastián Pérez Herrero

ISBN 978-84-945523-2-8

Capítulo 5. Máquinas de Estados Finitos

Importante

Este comportamiento ha sido declarado obsoleto a partir de la versión OTP 20. Si escribes tu código en versiones OTP 20 o superiores emplea mejor ***gen_statem***. Ver más en el Capítulo 6, ***Máquinas de Estados***.

Una máquina de estados es un sistema que permite presentar varios estados y las transiciones entre ellos. La máquina de estados finitos además limita el número de estados. Deben ser estados conocidos y no dinámicos.

Un ejemplo de máquina de estados es un reloj, que va cambiando la información de su hora cuando se produce un evento interno que le indica que debe hacerlo. También es una máquina de estados finitos un semáforo, he incluso podríamos considerar que una bombilla tiene dos estados (encendida y apagada).

Una de las grandes ventajas del Modelo Actor es la facilidad para implementar FSM (***Finite State Machines***, Máquinas de Estados Finitos) pudiendo partir de un esquema y desarrollar el código de forma transparente.

Los diagramas de estados son una gran herramienta en este capítulo y en el desarrollo de las FSM en general. Simplifica la visualización de los estados y los eventos que facilitan el cambio de estos estados. Los revisaremos a lo largo del capítulo.

1. Plantilla de un FSM

En principio necesitamos la plantilla de la que partir para escribir nuestro código. Las funciones publicadas en el comportamiento de FSM y las funciones a implementar en nuestro código.

Esta es nuestra plantilla:

```
-module(simplefsm).
-behaviour(gen_fsm).

-export([start_link/0]).

-export([
    init/1,
    init_state/2,
```

```
    init_state/3,
    handle_sync_event/4,
    handle_event/3,
    handle_info/3,
    terminate/2,
    code_change/4
]).

-record(state, {}).

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

init([]) ->
    {ok, init_state, #state{}}.

init_state(_Event, StateData) ->
    {next_state, init_state, StateData}.

init_state(_Event, _From, StateData) ->
    {reply, ok, init_state, StateData}.

handle_sync_event(_Event, StateName, _From, StateData) ->
    {reply, ok, StateName, StateData}.

handle_event(_Event, StateName, StateData) ->
    {next_state, StateName, StateData}.

handle_info(_Info, StateName, StateData) ->
    {next_state, StateName, StateData}.

terminate(_Reason, _StateName, _StateData) ->
    ok.

code_change(_OldVsn, StateName, StateData, _Extra) ->
    {ok, State}.
```

Nota

En esta plantilla aparecen las funciones `init_state/2` y `init_state/3`. Estas funciones son ejemplos de estados que habrá que implementar. Si ninguno de tus estados se llama así, puedes eliminarlas sin problema.

La plantilla es muy similar a la vista anteriormente para ***gen_server***. No obstante pueden verse algunas adiciones que veremos en detalle.

2. Eventos, Estados y Diagramas de Estados

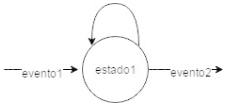
Una de las herramientas más importantes en la generación de una máquina de estados finitos son los diagramas de estados. Estos diagramas nos proporcionan en un vistazo todo el funcionamiento de una máquina de estados finitos.

Los **estados** en estos diagramas se representan con una circunferencia (o un círculo) con el nombre del estado escrito dentro. Idealmente todos tiene el mismo tamaño y el mismo color. Para facilitar la lectura en diagramas con muchos estados podemos cambiar el color de algunos de ellos para formar grupos visuales.

Los **iniciadores y terminadores** son líneas verticales u horizontales de cierto grosor. Indican el inicio y/o fin de la máquina de estados.

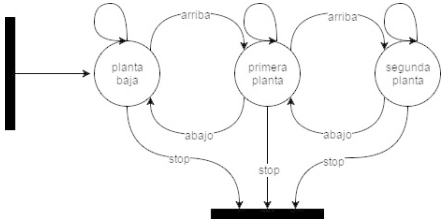
Los **eventos** son las líneas que permiten hacer el tránsito de un estado a otro. Estas flechas parten de un estado o iniciador y terminan en otro estado o finalizador. La punta de la flecha marca su dirección. A lo largo de su trayectoria se escribe el nombre del evento.

Los eventos también pueden hacer bucles, es decir iniciar y terminar en el mismo estado indicando que aunque se ha recibido el evento y se realiza una acción, no se modifica el estado.

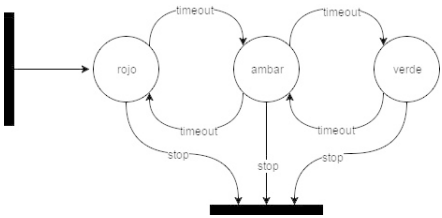


Los eventos generalmente son acciones que se realizan por una entrada de datos sobre la máquina de estados. Esto puede significar la recepción de un mensaje o incluso el disparo de un evento de tiempo.

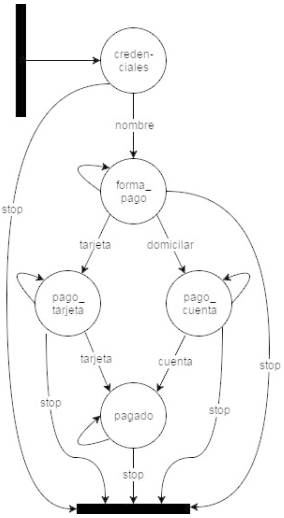
Para nuestros ejemplos vamos a dibujar sus diagramas de estados. El ejemplo del ascensor tendrá el siguiente diagrama:



El ejemplo del semáforo mostrará el siguiente diagrama:



Por último, el ejemplo de nuestro proceso de pago tendrá el siguiente diagrama:



Ahora veremos como implementarlos.

3. Iniciando la Máquina de Estados Finitos

Una máquina de estados tiene como base un servidor. El inicio es similar a ***gen_server*** sobretodo en el lanzamiento del proceso. La función que se encarga de esto es `gen_fsm:start_link/3-4` o `gen_fsm:start/3-4`. La primera de las opciones realiza un enlazado (***link***) del proceso mientras que la segunda de las opciones no lo hace.

Vemos a continuación la función `gen_fsm:start_link/3-4` para lanzar la máquina de estados:

```
-spec start_link(module(), args(), options()) -> result().
-spec start_link(servername(), module(), args(),
```

```
                options()) -> result().

-type servername() :: {local,name()} |
                      {global,globalname()} |
                      {via,module(),vianame()}.
-type name() :: atom().
-type globalname() :: term().
-type vianame() :: term().
-type args() :: term().
-type options() :: [option()].
-type option() :: {debug,dbgs()} |
                  {timeout,time()} |
                  {spawn_opt,sopts()}.
-type dbgs() :: [dbg()].
-type dbg() :: trace | log | statistics |
               {log_to_file,filename()} |
               {install, {function(), functionstate()}}.
-type sopts() :: [term()].

-type result() :: {ok,pid()} | ignore | {error,error()}.
-type error() :: {already_started,pid()} | term().
```

La función `gen_fsm:start/3-4` tiene los mismos parámetros. Sin embargo esta función no realiza un enlazado del proceso nuevo (`link/1`).

En nuestro ejemplo del ascensor podemos lanzar un proceso de máquina de estados de la siguiente forma:

```
start_link() ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE, [], []).
```

Esta ejecución nos permite generar un proceso con el nombre del módulo y acceder a él a través de ese nombre.

El proceso de pago podemos iniciarlo de forma anónima:

```
start_link() ->
    gen_fsm:start_link(?MODULE, [], []).
```

De esta forma podemos generar tantos como necesitemos. Por último, el proceso del semáforo lo iniciaremos pasándole parámetros:

```
start_link(TiempoVerde, TiempoRojo) ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE,
                       [TiempoVerde, TiempoRojo], []).
```

La función de inicio de la máquina de estados `init/1` tiene un parámetro. Recibirá lo enviado desde la función `gen_fsm:start_link/3-4`. El retorno incluirá los datos de estado y además el nombre del estado:

```
-spec init([term()]) -> result().
```

```
-type result() :: {ok, statename(), statedata()} |
                  {ok, statename(), statedata(), timeout()} |
                  {ok, statename(), statedata(), hibernate} |
                  {stop, reason()} |
                  ignore.
-type timeout() :: pos_integer() | infinity.
-type reason() :: term().
-type statename() :: atom().
-type statedata() :: term().
```

Las opciones relacionadas al servidor pueden verse en la Sección 2, "Iniciando el servidor" [33]. La aportación de ***gen_fsm*** es la agregación de ***statename()***. Este valor nos permite configurar el primer estado, el punto de entrada para el primer evento de nuestro sistema.

La implementación para nuestro ejemplo del ascensor es la siguiente:

```
init([]) ->
    {ok, planta_baja, #state{}}.
```

En nuestro semáforo la inicialización obtendrá dos parámetros. Definiremos antes un registro para el estado y la función de la siguiente forma:

```
-record(state, {
    tiempo_verde :: pos_integer(),
    tiempo_rojo :: pos_integer(),
    siguiente :: verde | rojo
}).

init([TiempoVerde, TiempoRojo]) ->
    {ok, rojo, #state{
        tiempo_verde = TiempoVerde,
        tiempo_rojo = TiempoRojo
    }, TiempoRojo}.
```

Iniciamos el semáforo en rojo y configuramos los tiempos de espera para cada uno de los colores. Al mismo tiempo empleamos el retorno de tiempo (***timeout***) para provocar una entrada de tiempo al acabar el tiempo de espera.

Por último, la implementación de la función `init/1` para nuestro sistema de pago. La toma de datos del sistema de pago consiste en varios pasos. Debemos almacenar esa información en el estado:

```
-record(state, {
    nombre :: string(),
    email :: string(),
    precio :: float(),
    tarjeta :: string()
    direccion :: string()
}).
```

```
init([]) ->
    {ok, identifica, #state{}}.
```

Comenzamos en el estado ***identifica*** y esperamos la llegada de los eventos.

4. Envío de eventos

Revisando los diagramas podemos ver los eventos. Son esas líneas que van de un estado a otro. Desde el iniciador hacia los eventos y de estos hasta el terminador. Los eventos pueden ser de varias formas:

Asíncronos

Son los eventos enviados a la máquina de estados y de los que no esperamos una respuesta. Equivalente al envío de información del servidor (***cast***).

Síncronos

Eventos enviados a la máquina de estados y de los que esperamos una respuesta. Equivalente a la llamada del servidor (***call***).

Específicos

Enviamos el evento a la máquina de estados esperando que sea atendido únicamente por el estado activo.

Generales

El evento se envía a la máquina de estados para que sea atendido por la implementación general común a todos los estados.

4.1. Eventos asíncronos y específicos

En principio vamos a centrarnos en los eventos asíncronos. Estos son los eventos más empleados. Los eventos asíncronos son disparados a través de la función `gen_fsm:send_event/2`. Vemos la especificación de esta función:

```
-spec send_event(server_ref(), event()) -> ok.

-type process_name() :: atom() | pid().
-type global_name() :: term().
-type via_name() :: term().
-type server_ref() :: process_name() |
                      {process_name(), node()} |
                      {global, global_name()} |
                      {via, module(), via_name()} | pid().
-type event() :: term().
```

Nuestro ejemplo del ascensor tendrá dos botones que podemos presionar para realizar las acciones de cambio de planta. Implementamos estos botones de la siguiente forma:

```
boton_arriba() ->
    gen_fsm:send_event(?MODULE, arriba).

boton_abajo() ->
    gen_fsm:send_event(?MODULE, abajo).
```

Cada uno envía un evento diferente a la máquina de estados y según el estado nuestra máquina reaccionará de una forma u otra.

4.2. Eventos síncronos y específicos

El evento síncrono envía la información a la máquina de estados y espera una respuesta. La función llamante queda bloqueada hasta recibir la respuesta o se agote el tiempo de espera. La función a emplear es `gen_fsm:sync_send_event/2-3` y se define así:

```
-spec sync_send_event(server_ref(), event()) -> result().
-spec sync_send_event(server_ref(), event(), timeout()) ->
 result().

-type process_name() :: atom() | pid().
-type global_name() :: term().
-type via_name() :: term().
-type server_ref() :: process_name() |
                      {process_name(), node()} |
                      {global, global_name()} |
                      {via, module(), via_name()} | pid().
-type event() :: term().
-type timeout() :: pos_integer() | infinity.
-type result() :: term().
```

El proceso de pago requiere de eventos síncronos para detectar si el paso realizado es correcto o no. Tendrá las siguientes peticiones para generar envíos a la máquina de estados:

```
da_nombre(PID, Nombre) ->
    gen_fsm:sync_send_event(PID, {nombre, Nombre}).

da_forma_pago(PID, FormaPago) ->
    gen_fsm:sync_send_event(PID, {forma_pago, FormaPago}).

da_tarjeta(PID, Tarjeta) ->
    gen_fsm:sync_send_event(PID, {tarjeta, Tarjeta}).

da_cuenta(PID, Cuenta) ->
    gen_fsm:sync_send_event(PID, {cuenta, Cuenta}).
```

El evento es enviado a la máquina de estados y obtendremos una respuesta correcta o errónea dependiendo del estado.

4.3. Eventos generales

Los dos tipos de eventos vistos hasta el momento son específicos. Serán atendidos por el estado activo. No obstante, quizás necesitemos implementar una funcionalidad que esté presente independientemente del estado. Sea asíncrona o síncrona. Esto lo podemos conseguir con las funciones `gen_fsm:send_all_state_event/2` y `gen_fsm:sync_send_all_state_event/2-3`. La definición de estas funciones es la siguiente:

```
-spec send_all_state_event(server_ref(), event()) -> ok.
-spec sync_send_all_state_event(server_ref(), event()) ->
 result().
-spec sync_send_all_state_event(server_ref(), event(),
 timeout()) -> result().

-type process_name() :: atom() | pid().
-type global_name() :: term().
-type via_name() :: term().
-type server_ref() :: process_name() |
                      {process_name(), node()} |
                      {global, global_name()} |
                      {via, module(), via_name()} | pid().
-type event() :: term().
-type result() :: term().
```

La implementación para el ejemplo de nuestro semáforo puede ser la siguiente:

```
ver_semaforo() ->
    gen_fsm:sync_send_all_state_event(?MODUEL, ver_semaforo).
```

La ejecución nos dirá en qué estado está el semáforo (rojo, ámbar o verde).

En cualquiera de nuestros ejemplos podemos implementar el evento general asíncrono. La parada de la máquina de estados es algo genérico a cada uno de ellos:

```
stop() ->
    gen_fsm:send_all_state_event(?MODULE, stop).
```

5. Transición de Estado

La máquina de estados finitos reacciona al recibir un evento ejecutando una de las posibles implementaciones. Depende del tipo de evento se selecciona una implementación u otra. Tenemos cuatro posibilidades: evento asíncrono específico, evento asíncrono general, evento síncrono específico y evento síncrono general.

Al finalizar el evento podemos especificar si existirá una transición de estado, una finalización de la máquina de estados o si nos mantenemos en el mismo estado.

5.1. Evento asíncrono específico

Este evento se forma a través del uso del nombre de estado como nombre de la función a ser llamada. Si el estado es ***rojo*** la función a ser llamada en este caso es `rojo/2`. Veamos la forma genérica:

```
-spec StateName(event(), state()) -> result().

% StateName :: atom()
-type event() :: term().
-type state() :: term().
-type state_name() :: atom().
-type timeout() :: pos_integer().
-type reason() :: term().
-type result() ::
    {next_state, state_name(), state()} |
    {next_state, state_name(), state(), timeout()} |
    {next_state, state_name(), state(), hibernate} |
    {stop, reason(), state()}.
```

La forma general nos proporciona la base para la implementación. Los parámetros a recibir y los retornos de la función.

Para nuestro ejemplo del ascensor implementaremos las siguientes funciones de estados:

```
planta_baja(up, StateData) ->
    io:format("subiendo al primer piso~n", []),
    {next_state, primera_planta, StateData};
planta_baja(down, StateData) ->
    io:format("beeeep! no se puede bajar~n", []),
    {next_state, planta_baja, StateData}.

primera_planta(up, StateData) ->
    io:format("subiendo al segundo piso~n", []),
    {next_state, segunda_planta, StateData};
primera_planta(down, StateData) ->
    io:format("bajando a planta baja~n", []),
    {next_state, planta_baja, StateData}.

segunda_planta(up, StateData) ->
    io:format("beeeep! no se puede subir~n", []),
    {next_state, segunda_planta, StateData};
segunda_planta(down, StateData) ->
    io:format("bajando al primer piso~n", []),
    {next_state, primera_planta, StateData}.
```

La implementación muestra la misma forma del diagrama de estados. Tres estados: ***planta_baja***, ***primera_planta*** y ***segunda_planta***. Según el evento recibido podemos realizar una acción y cambiar de estado o continuar en el mismo estado.

5.2. Evento síncrono específico

Formamos el evento síncrono con el nombre del estado y tres parámetros: el evento recibido, quién lo envía y los datos del estado del proceso. La especificación de la función tiene esta forma:

```
-spec StateName(event(), from(), state()) -> result().

% StateName :: atom()
-type event() :: term().
-type from() :: {pid(), reference()}.
-type state() :: term().
-type state_name() :: atom().
-type timeout() :: pos_integer().
-type reason() :: term().
-type reply() :: term().
-type result() ::
    {reply, reply(), state_name(), state()} |
    {reply, reply(), state_name(), state(), timeout()} |
    {reply, reply(), state_name(), state(), hibernate} |
    {next_state, state_name(), state()} |
    {next_state, state_name(), state(), timeout()} |
    {next_state, state_name(), state(), hibernate} |
    {stop, reason(), state()}.
```

Recuerda que cambiamos ***StateName*** por el nombre de nuestros estados agregando tantas funciones como estados tengamos.

Nota

En los códigos de ejemplo puedes ver la implementación de estas funciones solo aparecen cuando se emplean. En el ejemplo de pago tenemos su implementación pero no en los otros ejemplos. La implementación de estas funciones es opcional. No obstante si envías un evento síncrono y la función no existe tu proceso se detendrá debido al error.

La implementación en nuestro ejemplo de pago de los estados síncronos es como sigue:

```
credenciales({nombre, Nombre}, _From, State) ->
    {reply, ok, forma_pago, State#state{nombre=Nombre}};
credenciales(_Msg, _From, State) ->
    {reply, {error, "necesitamos nombre!"},
     credenciales, State}.

forma_pago({forma_pago, tarjeta}, _From, State) ->
    {reply, ok, pago_tarjeta, State};
forma_pago({forma_pago, domiciliar}, _From, State) ->
    {reply, ok, pago_domiciliar, State};
forma_pago(_Msg, _From, State) ->
    {reply, {error, "forma de pago: tarjeta o domiciliar"},
     forma_pago, State}.
```

```
pago_tarjeta({tarjeta, Tarjeta}, _From, State) ->
    {reply, {ok, pagado}, pagado,
     State#state{tarjeta=Tarjeta},
     hibernate};
pago_tarjeta(_Msg, _From, State) ->
    {reply, {error, "necesitamos tarjeta"}, pago_tarjeta,
     State}.

pago_cuenta({cuenta, Cuenta}, _From, State) ->
    {reply, {ok, pagado}, pagado, State#state{cuenta=Cuenta},
     hibernate};
pago_cuenta(_Msg, _From, State) ->
    {reply, {error, "necesitamos cuenta"}, pago_cuenta,
     State}.

pagado(_Event, _From, State) ->
    {reply, {error, ya_pagado}, State, hibernate}.
```

Puedes ver que hay cuatro bloques diferenciados de funciones uno para cada estado. Usamos concordancia para distinguir entre las funciones llamadas con datos correctos y el caso generar que retornará un error.

Estando en el estado ***credenciales*** si enviamos un evento ***{tarjeta, "12345678"}*** obtendremos un error indicándonos el tipo de dato que se espera en ese estado.

5.3. Evento asíncrono general

Hay casos en los que un evento es independiente del estado. En este caso empleamos la implementación de la función `handle_event/3`. Esta función es parecida a las funciones de eventos específicos pero el estado es enviado como parámetro en lugar de usarlo como nombre de la función. La función debe ser implementada en nuestros ejemplos y tiene la siguiente forma:

```
-spec handle_event(event(), statename(), statedata()) ->
 result().

% StateName :: atom()
-type event() :: term().
-type statedata() :: term().
-type state_name() :: atom().
-type timeout() :: pos_integer().
-type reason() :: term().
-type result() ::
    {next_state, state_name(), state()} |
    {next_state, state_name(), state(), timeout()} |
    {next_state, state_name(), state(), hibernate} |
    {stop, reason(), state()}.
```

Esta función es llamada cuando empleamos la función `gen_fsm:send_all_state_event/2`. En nuestros ejemplos hemos implementado esta función para realizar la parada de la FSM. Podemos ver la implementación específica aquí:

```
handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.
```

En este caso el nombre del estado no es importante para nosotros. Sin embargo si queremos permitir la parada solo en determinados estados, podemos agregar guardas para filtrar esos estados y una implementación general para ignorar los que no concuerden.

5.4. Evento síncrono general

La implementación del evento síncrono general se realiza a través de la llamada de la función `gen_fsm:sync_send_all_state_event/2`. Esta función no depende del estado y siempre llama a la implementación de la retrollamada `handle_sync_event/4` definida en nuestro código. La especificación de esta función es así:

```
-spec handle_sync_event(event(), from(), statename(),
 statedata()) -> result().

% StateName :: atom()
-type event() :: term().
-type from() :: {pid(), reference()}.
-type statedata() :: term().
-type state_name() :: atom().
-type timeout() :: pos_integer().
-type reason() :: term().
-type reply() :: term().
-type result() ::
    {reply, reply(), state_name(), state()} |
    {reply, reply(), state_name(), state(), timeout()} |
    {reply, reply(), state_name(), state(), hibernate} |
    {next_state, state_name(), state()} |
    {next_state, state_name(), state(), timeout()} |
    {next_state, state_name(), state(), hibernate} |
    {stop, reason(), state()}.
```

Esta función recibe un evento con la información del nombre del estado y los datos del estado del proceso, además de la información de quién realizó la llamada al proceso. El retorno puede ser un cambio de estado e ignorar la respuesta, una respuesta con cambio de estado o la detención de la máquina de estados.

Nota

Al igual que en el caso de ***gen_server*** puedes enviar a otro proceso la información de ***From*** y con `gen_server:reply/2` realizar el envío de la respuesta al proceso llamante.

6. Eventos de tiempo

El semáforo no tendrá eventos entrantes por el usuario, solo de tiempo.

La implementación de estados es la siguiente:

```
rojo(timeout, State) ->
    {next_state, ambar, State#state{siguiente=verde},
     ?TIEMPO_EN_AMBAR}.

ambar(timeout, #state{siguiente=verde}=State) ->
    {next_state, verde, State, State#state.tiempo_verde};
ambar(timeout, #state{siguiente=rojo}=State) ->
    {next_state, rojo, State, State#state.tiempo_rojo}.

verde(timeout, State) ->
    {next_state, ambar, State#state{siguiente=rojo},
     ?TIEMPO_EN_AMBAR}.
```

En este caso los eventos son todos ***timeout***. Según el estado cambiamos a un color u otro. En el retorno vemos el uso del cuarto elemento de la tupla para realizar un evento de tipo ***timeout***.

No obstante, este timeout tiene un problema. Si enviásemos un evento o información sin formato al proceso el contador se reiniciaría. Tardaría más tiempo en realizar la transición de estado. Esto podemos resolverlo con el uso de la función `gen_fsm:send_event_after/2`. Esta función solo la podemos emplear dentro del proceso de la FSM debido a que los dos parámetros que acepta son el tiempo antes de enviar el evento y el evento a ser enviado. El evento se enviará al proceso llamante. La especificación de la función es la siguiente:

```
-spec gen_fsm:send_event_after(time(), event()) ->
 reference().

-type time() :: pos_integer().
-type event() :: term().
```

Podemos reimplementar nuestro ejemplo del semáforo para emplear solo este tipo de evento en lugar de emplear el parámetro ***timeout*** del retorno de la función. Como segunda versión obtenemos este código para la función de inicio:

```
init([TiempoVerde, TiempoRojo]) ->
    gen_fsm:send_event_after(TiempoRojo, timeout),
    {ok, rojo, #state{
        tiempo_verde = TiempoVerde,
        tiempo_rojo = TiempoRojo
    }}.
```

Los eventos también cambian para emplear esta función. Tenemos que modificarlos de la siguiente forma:

```
rojo(timeout, State) ->
    gen_fsm:send_event_after(?TIEMPO_AMBAR, timeout),
```

```
    {next_state, ambar, State#state{siguiente=verde}}.

ambar(timeout, #state{siguiente=verde}=State) ->
    gen_fsm:send_event_after(State#state.tiempo_verde,
                             timeout),
    {next_state, verde, State};
ambar(timeout, #state{siguiente=rojo}=State) ->
    gen_fsm:send_event_after(State#state.tiempo_rojo,
                             timeout),
    {next_state, rojo, State}.

verde(timeout, State) ->
    gen_fsm:send_event_after(?TIEMPO_AMBAR, timeout),
    {next_state, ambar, State#state{siguiente=rojo}}.
```

En la sección de cambio de código en caliente veremos cómo realizar el cambio entre una versión y otra.

7. Hibernación de la FSM

En nuestro ejemplo de pago hibernamos el proceso cuando el pago se ha terminado. Esto nos permite seguir obteniendo la información del proceso y al mismo tiempo utilizar menos memoria y CPU.

No obstante el proceso debe ser terminado explícitamente para poder salir de la memoria del sistema completamente. Mantenerlo hibernado nos garantiza un empleo de memoria inferior y nada de uso de la CPU. Cada vez que solicitemos información tendremos que enviarlo a hibernación también. Igualmente podríamos programar un evento de tiempo para realizar la parada completa del proceso. Lo dejo para vuestra experimentación. A través de la función `erlang:process_info/1` podéis obtener información del proceso e ir probando.

8. Información no esperada o sin formato

En nuestros ejemplos no hemos hecho tratamiento de información sin formato. Esta es la información que se recibe a través de la implementación de la función `handle_info/3` y tiene esta definición:

```
-spec handle_info(info(), state_name(), state_data()) ->
 result().

-type info() :: term().
-type statedata() :: term().
-type state_name() :: atom().
-type timeout() :: pos_integer().
-type reason() :: term().
-type result() ::
    {next_state, state_name(), state()} |
    {next_state, state_name(), state(), timeout()} |
    {next_state, state_name(), state(), hibernate} |
    {stop, reason(), state()}.
```

Para obtener información de envío de información directamente al proceso o a través de un puerto de comunicación de red es necesario el uso de `handle_info/3` pero fuera de esos casos no aconsejo su uso. La razón es que los otros métodos son más explicativos acerca del origen de la información y son más fáciles de trazar en caso de errores.

9. Finalizando la Máquina de Estados Finitos

Una vez detenemos la FSM se ejecuta la función `terminate/3`. Esta función tiene la siguiente definición:

```
-spec terminate(reason(), state_name(), state_data()) ->
 no_return().

-type state_data() :: term().
-type state_name() :: atom().
-type reason() :: term().
```

El retorno de la función no es tenido en cuenta. El proceso muere justo tras su ejecución. La implementación por defecto para la mayoría de nuestros ejemplos es la siguiente:

```
terminate(_Reason, StateName, _StateData) ->
    ok.
```

Para el ejemplo de pago hemos implementado otra forma de terminar:

```
terminate(_Reason, pagado, #state{forma_pago=tarjeta}) ->
    io:format("~p Nombre: ~s~nTarjeta: ~s~nPagado.~n",
              [self(), State#state.nombre,
               State#state.tarjeta]),
    ok;
terminate(_Reason, pagado, #state{forma_pago=cuenta}) ->
    io:format("~p Nombre: ~s~nCuenta: ~s~nPagado.~n",
              [self(), State#state.nombre,
               State#state.cuenta]),
    ok;
terminate(_Reason, _StateName, _StateData) ->
    io:format("~p No pagado.~n", [self()]),
    ok.
```

Dependiendo del nombre y los datos del estado mostraremos una información u otra justo al finalizar la ejecución del proceso.

Importante

Si el proceso finaliza debido a un error o excepción no ejecutará esta función y no se mostrará la información por pantalla.

10. Cambio de código en caliente

En nuestro ejemplo del semáforo vimos que teníamos dos formas de implementar el evento de tiempo de espera agotado (o ***timeout***). Teniendo la primera versión con el tiempo de espera enviado como retorno en la función y una segunda versión con el envió del evento de tiempo a través de la función `gen_fsm:send_event_after/2`. Vamos a actualizar nuestro código sin detener el sistema.

Un requisito previo es la especificación de las versiones de los módulos. Para simplificar. En las primeras líneas podemos agregar el mandato de preprocesador de esta forma:

```
-module(semaforo).
-author('manuel@altenwald.com').
-vsn(1).
```

En el fichero de la segunda versión tenemos que agregar lo mismo pero sumando un número a la versión:

```
-module(semaforo).
-author('manuel@altenwald.com').
-vsn(2).
```

Los cambios en el código los comentamos en la sección de eventos de tiempo. El código correspondiente a la actualización para el segundo fichero debe implementarse en la función `code_change/4`. Esta función tiene la siguiente definición:

```
-spec code_change(old_vsn(), state_name(), state_data(),
 extra()) -> {ok, state_name(), state_data()} | {error,
 reason()}.

-type old_vsn() :: vsn() | {down, vsn()}.
-type vsn() :: term().
-type state_name() :: atom().
-type state_data() :: term().
-type reason() :: term().
-type extra() :: term().
```

La función debe determinar qué hacer para actualizar el estado de una versión antigua a una nueva. El primer parámetro nos ayudará a saber de qué versión partimos. Igualmente también puede realizarse una desactualización o vuelta atrás (***downgrade***) forzando a volver a un código antiguo. Esta acción se indica en el primer parámetro con la recepción de ***{down, Vsn}*** donde la variable ***Vsn*** será la versión a la que se vuelve.

Obtenemos como los parámetros dos y tres, el nombre y datos del estado respectivamente y como último parámetro opciones extra que pueden

enviarse al actualizar el código. Útil si hay información no existente en la versión antigua y queramos enviarla en el momento de actualizar sin ligarla al código en sí.

La implementación del cambio de código para el semáforo es la siguiente:

```
code_change(_OldVsn, rojo,
            #state{tiempo_rojo=TiempoRojo}=State,
            _Extra) ->
    gen_fsm:send_event_after(TiempoRojo, timeout),
    {ok, rojo, State};
code_change(_OldVsn, verde,
            #state{tiempo_rojo=TiempoVerde}=State,
            _Extra) ->
    gen_fsm:send_event_after(TiempoVerde, timeout),
    {ok, verde, State};
code_change(_OldVsn, ambar, State, _Extra) ->
    gen_fsm:send_event_after(?TIEMPO_AMBAR, timeout),
    {ok, ambar, State}.
```

Según el estado elegimos un tiempo de espera y lanzamos el temporizador para asegurar el cambio de estado justo cuando pase ese tiempo.

Abrimos una consola y compilamos el primer código. Lanzamos el proceso:

```
> c("semaforo_v1/semaforo.erl").
{ok,semaforo}
> semaforo:start_link(5000, 5000).
{ok,<0.63.0>}
> semaforo:ver_semaforo().
rojo
> semaforo:ver_semaforo().
rojo
```

Ejecutando cada sentencia antes de que pasen los 5 segundos para el cambio vemos que siempre obtenemos el valor ***rojo***. No cambia porque nuestra petición de ver el semáforo lo impide.

Vamos a actualizar nuestro código:

```
> sys:suspend(semaforo).
ok
> c("semaforo_v2/semaforo.erl").
{ok,semaforo}
> sys:change_code(semaforo, semaforo, 1, []).
ok
> sys:resume(semaforo).
ok
> semaforo:ver_semaforo().
rojo
> semaforo:ver_semaforo().
ambar
```

Ahora sí cambia. Las peticiones de datos no afectan al temporizador.

11. Obteniendo información de la Máquina de Estados Finitos

En Sección 10, "Obteniendo información del servidor" hablamos de cómo mostrar la información de un servidor. A todos los efectos una FSM es un servidor. No obstante la información interna cambia sutilmente al integrar el nombre del estado.

La especificación de la función `format_status/2` es exactamente igual. Podemos ver un ejemplo del volcado por defecto de la información de una FSM aquí:

```
> sys:get_status(ascensor).
{status,<0.47.0>,
        {module,gen_fsm},
        [[{'$ancestors',[<0.31.0>]},
          {'$initial_call',{ascensor,init,1}}],
         running,<0.31.0>,[],
         [{header,"Status for state machine ascensor"},
          {data,[{"Status",running},
                 {"Parent",<0.31.0>},
                 {"Logged events",[]},
                 {"StateName",planta_baja}]},
          {data,[{"StateData",{state}}]}]]}
```

Podemos ver que hay una sección que nos dice su ***StateName*** y la sección de ***StateData*** que tiene el estado vacío.

12. Ejemplo del Ascensor

A lo largo del capítulo hemos visto la implementación del ascensor por partes. Ahora vamos a ver el código completo de esta FSM y probaremos su ejecución.

El código completo es el siguiente:

```
-module(ascensor).
-author('manuel@altenwald.com').

-behaviour(gen_fsm).

-export([
    start_link/0,
    stop/0,
    boton_arriba/0,
    boton_abajo/0
]).

-export([
    init/1,
    planta_baja/2,
```

```
    primera_planta/2,
    segunda_planta/2,
    handle_sync_event/4,
    handle_event/3,
    handle_info/3,
    terminate/3,
    code_change/4
]).

-record(state, {}).

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

stop() ->
    gen_fsm:send_all_state_event(?MODULE, stop).

boton_arriba() ->
    gen_fsm:send_event(?MODULE, up).

boton_abajo() ->
    gen_fsm:send_event(?MODULE, down).

init([]) ->
    io:format("iniciamos en el primer piso~n", []),
    {ok, planta_baja, #state{}}.

planta_baja(up, StateData) ->
    io:format("subiendo al primer piso~n", []),
    {next_state, primera_planta, StateData};
planta_baja(down, StateData) ->
    io:format("beeeep! no se puede bajar~n", []),
    {next_state, planta_baja, StateData}.

primera_planta(up, StateData) ->
    io:format("subiendo al segundo piso~n", []),
    {next_state, segunda_planta, StateData};
primera_planta(down, StateData) ->
    io:format("bajando a planta baja~n", []),
    {next_state, planta_baja, StateData}.

segunda_planta(up, StateData) ->
    io:format("beeeep! no se puede subir~n", []),
    {next_state, segunda_planta, StateData};
segunda_planta(down, StateData) ->
    io:format("bajando al primer piso~n", []),
    {next_state, primera_planta, StateData}.

handle_sync_event(_Event, StateName, _From, StateData) ->
    {reply, ok, StateName, StateData}.

handle_event(_Event, StateName, StateData) ->
    {next_state, StateName, StateData}.

handle_info(_Info, StateName, StateData) ->
    {next_state, StateName, StateData}.

terminate(_Reason, _StateName, _StateData) ->
    ok.

code_change(_OldVsn, StateName, StateData, _Extra) ->
```

```
    {ok, StateName, StateData}.
```

Entramos en una consola y lo probamos de la siguiente forma:

```
> c(ascensor).
{ok,ascensor}
> ascensor:start_link().
iniciamos en el primer piso
{ok,<0.38.0>}
> ascensor:boton_arriba().
subiendo al primer piso
ok
> ascensor:boton_arriba().
subiendo al segundo piso
ok
> ascensor:boton_arriba().
beeeep! no se puede subir
ok
> ascensor:boton_abajo().
bajando al primer piso
ok
> ascensor:boton_abajo().
bajando a planta baja
ok
> ascensor:boton_abajo().
beeeep! no se puede bajar
ok
> ascensor:stop().
ok
```

El ascensor se inicia como proceso y podemos hacer que suba y baje usando las funciones correspondientes. Como último paso podemos detenerlo. La implementación de nuestro ascensor funciona correctamente.

13. Ejemplo del Semáforo

Para el semáforo tenemos dos implementaciones. Vimos en el apartado de cambio de código en caliente cómo realizar la actualización de uno a otro y aquí tenemos la segunda versión de nuestro código:

```
-module(semaforo).
-author('manuel@altenwald.com').
-vsn(2).

-behaviour(gen_fsm).

-export([
    start_link/2,
    stop/0,
    ver_semaforo/0
]).

-export([
    init/1,
    rojo/2,
    ambar/2,
```

```
    verde/2,
    handle_sync_event/4,
    handle_event/3,
    handle_info/3,
    terminate/3,
    code_change/4
]).

-define(TIEMPO_EN_AMBAR, 1000). %% 1 segundo

-record(state, {
    tiempo_verde :: pos_integer(),
    tiempo_rojo :: pos_integer(),
    siguiente :: verde | rojo
}).

start_link(TiempoVerde, TiempoRojo) ->
    gen_fsm:start_link({local, ?MODULE}, ?MODULE,
                       [TiempoVerde, TiempoRojo], []).

stop() ->
    gen_fsm:send_all_state_event(?MODULE, stop).

ver_semaforo() ->
    gen_fsm:sync_send_all_state_event(?MODULE, ver_semaforo).

init([TiempoVerde, TiempoRojo]) ->
    gen_fsm:send_event_after(TiempoRojo, timeout),
    {ok, rojo, #state{
        tiempo_verde = TiempoVerde,
        tiempo_rojo = TiempoRojo
    }}.

rojo(timeout, State) ->
    gen_fsm:send_event_after(?TIEMPO_EN_AMBAR, timeout),
    {next_state, ambar, State#state{siguiente=verde}}.

ambar(timeout, #state{siguiente=verde}=State) ->
    gen_fsm:send_event_after(State#state.tiempo_verde,
 timeout),
    {next_state, verde, State};
ambar(timeout, #state{siguiente=rojo}=State) ->
    gen_fsm:send_event_after(State#state.tiempo_rojo,
 timeout),
    {next_state, rojo, State}.

verde(timeout, State) ->
    gen_fsm:send_event_after(?TIEMPO_EN_AMBAR, timeout),
    {next_state, ambar, State#state{siguiente=rojo}}.

handle_sync_event(ver_semaforo, _From, StateName, StateData) -
>
    {reply, StateName, StateName, StateData}.

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.

handle_info(_Info, StateName, StateData) ->
    {next_state, StateName, StateData}.

terminate(_Reason, _StateName, _StateData) ->
    ok.
```

```
code_change(_OldVsn, rojo,
 #state{tiempo_rojo=TiempoRojo}=State, _Extra) ->
    gen_fsm:send_event_after(TiempoRojo, timeout),
    {ok, State};
code_change(_OldVsn, verde,
 #state{tiempo_rojo=TiempoVerde}=State, _Extra) ->
    gen_fsm:send_event_after(TiempoVerde, timeout),
    {ok, State};
code_change(_OldVsn, ambar, State, _Extra) ->
    gen_fsm:send_event_after(?TIEMPO_EN_AMBAR, timeout),
    {ok, State}.
```

Entramos en una consola y probamos una ejecución:

```
> semaforo:start_link(2000, 2000).
{ok,<0.76.0>}
> semaforo:ver_semaforo().
rojo
> semaforo:ver_semaforo().
ambar
> semaforo:ver_semaforo().
verde
> semaforo:ver_semaforo().
rojo
> semaforo:stop().
ok
```

Los cambios se suceden correctamente como ya habíamos visto igualmente en la prueba del cambio de código en caliente.

14. Ejemplo de Pago

En las secciones anteriores hemos visto cómo progresaba nuestro código para implementar el diagrama de estados de pago. Este ejemplo es un poco más largo que los anteriores porque dispone de más estados y una bifurcación en uno de los estados.

No obstante este sistema es lineal. No vuelve a estados anteriores y finaliza tras realizar todas las transiciones programadas. Esto lo hace el más fácil de seguir.

Su código completo puede verse a continuación:

```
-module(pago).
-author('manuel@altenwald.com').

-behaviour(gen_fsm).

-export([
    start_link/0,
    stop/1,
    da_nombre/2,
    da_forma_pago/2,
    da_tarjeta/2,
    da_cuenta/2,
```

```
    obtiene_info/1
]).

-export([
    init/1,
    credenciales/3,
    forma_pago/3,
    pago_tarjeta/3,
    pago_cuenta/3,
    pagado/3,
    handle_sync_event/4,
    handle_event/3,
    handle_info/3,
    terminate/3,
    code_change/4
]).

-type forma_pago() :: tarjeta | domiciliar.

-record(state, {
 nombre :: string(),
 forma_pago :: forma_pago(),
 tarjeta :: string(),
 cuenta :: string()
}).

start_link() ->
    gen_fsm:start_link(?MODULE, [], []).

stop(Name) ->
    gen_fsm:send_all_state_event(Name, stop).

da_nombre(PID, Nombre) ->
    gen_fsm:sync_send_event(PID, {nombre, Nombre}).

da_forma_pago(PID, FormaPago) ->
    gen_fsm:sync_send_event(PID, {forma_pago, FormaPago}).

da_tarjeta(PID, Tarjeta) ->
    gen_fsm:sync_send_event(PID, {tarjeta, Tarjeta}).

da_cuenta(PID, Cuenta) ->
    gen_fsm:sync_send_event(PID, {cuenta, Cuenta}).

obtiene_info(PID) ->
    gen_fsm:sync_send_all_state_event(PID, info).

init([]) ->
    {ok, credenciales, #state{}}.

credenciales({nombre, Nombre}, _From, State) ->
 {reply, ok, forma_pago, State#state{nombre=Nombre}};
credenciales(_Msg, _From, State) ->
 {reply, {error, "necesitamos nombre!"}, credenciales, State}.

forma_pago({forma_pago, tarjeta}, _From, State) ->
 {reply, ok, pago_tarjeta, State#state{forma_pago=tarjeta}};
forma_pago({forma_pago, domiciliar}, _From, State) ->
 {reply, ok, pago_cuenta, State#state{forma_pago=domiciliar}};
forma_pago(_Msg, _From, State) ->
 {reply, {error, "forma de pago: tarjeta o domiciliar"},
 forma_pago, State}.
```

```
pago_tarjeta({tarjeta, Tarjeta}, _From, State) ->
 {reply, {ok, pagado}, pagado, State#state{tarjeta=Tarjeta},
 hibernate};
pago_tarjeta(_Msg, _From, State) ->
 {reply, {error, "necesitamos tarjeta"}, pago_tarjeta, State}.

pago_cuenta({cuenta, Cuenta}, _From, State) ->
 {reply, {ok, pagado}, pagado, State#state{cuenta=Cuenta},
 hibernate};
pago_cuenta(_Msg, _From, State) ->
 {reply, {error, "necesitamos cuenta"}, pago_cuenta, State}.

pagado(_Event, _From, State) ->
    {reply, {error, ya_pagado}, State, hibernate}.

handle_sync_event(info, _From, StateName, StateData) ->
    {reply, {StateName, StateData}, StateName, StateData}.

handle_event(stop, _StateName, StateData) ->
    {stop, normal, StateData}.

handle_info(_Info, StateName, StateData) ->
    {next_state, StateName, StateData}.

terminate(_Reason, pagado, #state{forma_pago=tarjeta}=State) -
>
    io:format("~p Nombre: ~s~nTarjeta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.tarjeta]),
    ok;
terminate(_Reason, pagado,
 #state{forma_pago=domiciliar}=State) ->
    io:format("~p Nombre: ~s~nCuenta: ~s~nPagado.~n",
              [self(), State#state.nombre,
 State#state.cuenta]),
    ok;
terminate(_Reason, _StateName, _StateData) ->
    io:format("~p No pagado.~n", [self()]),
    ok.

code_change(_OldVsn, _StateName, StateData, _Extra) ->
    {ok, StateData}.
```

Para ejecutarlo vamos a una consola y seguimos estos pasos:

```
> {ok, PID} = pago:start_link().
{ok,<0.33.0>}
> pago:obtiene_info(PID).
{credenciales,{state,undefined,undefined,undefined,
                     undefined}}
> pago:da_nombre(PID, "Manuel").
ok
> pago:obtiene_info(PID).
{forma_pago,{state,"Manuel",undefined,undefined,undefined}}
> pago:da_forma_pago(PID, cuenta).
{error,"forma de pago: tarjeta o domiciliar"}
> pago:da_forma_pago(PID, domiciliar).
ok
> pago:obtiene_info(PID).
{pago_cuenta,{state,"Manuel",domiciliar,undefined,
```

```
                    undefined}}
> pago:da_tarjeta(PID, "1234").
{error,"necesitamos cuenta"}
> pago:da_cuenta(PID, "1234").
{ok,pagado}
> pago:obtiene_info(PID).
{pagado,{state,"Manuel",domiciliar,undefined,"1234"}}
> pago:stop(PID).
<0.33.0> Nombre: Manuel
Cuenta: 1234
Pagado.
ok
```

Podemos ver la progresión de las peticiones de pago hasta llegar a la parada del proceso y obtener toda la información. Podemos probar a detener antes el proceso y ver el estado en el que queda la operación. Puedes realizar todas las pruebas y cambios que desees.